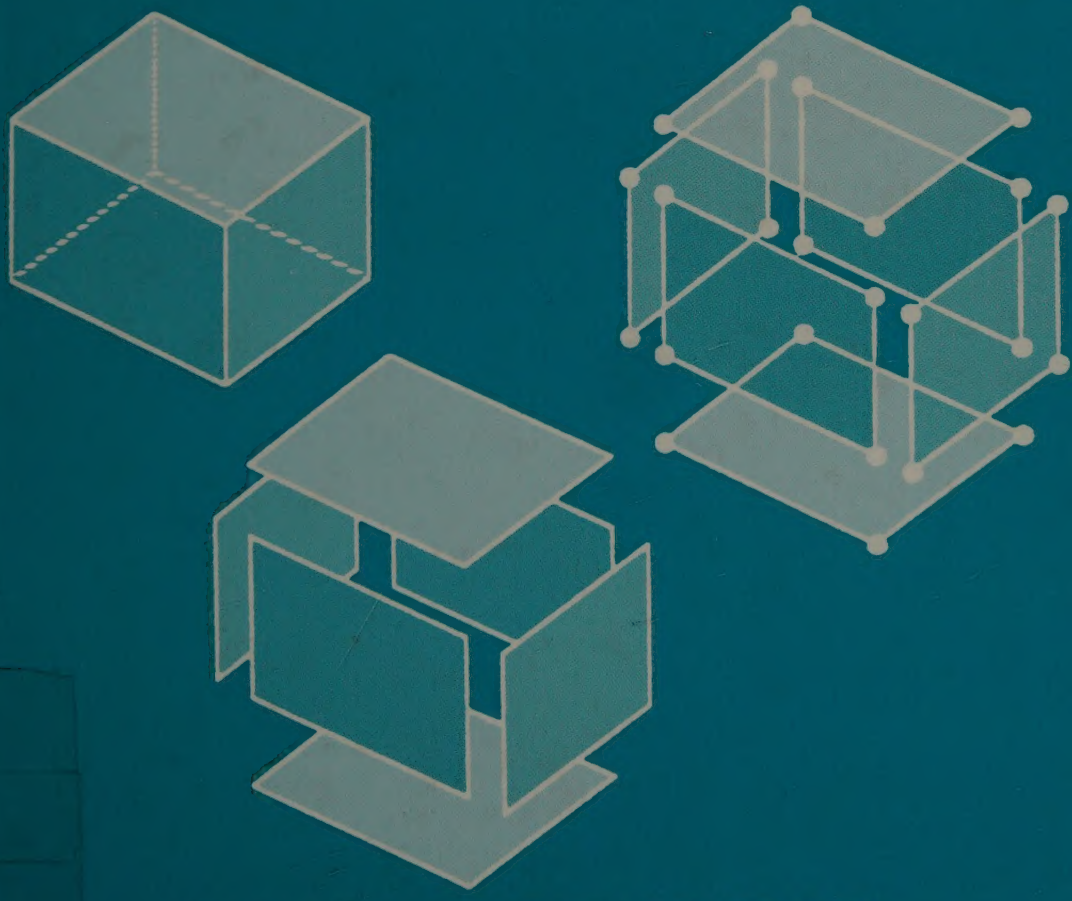
# AN INTRODUCTION TO SOLID MODELING

MARTTI MÄNTYLÄ



COMPUTER SCIENCE PRESS

ENGINEERING AND SCIENCE LIBRARY,
P.C.L., 115 NEW CAVENDISH ST., LONDON, W1M 8JS

pcl

**Polytechnic of Central London Library Services**

**Engineering and Science Library**
115 New Cavendish Street, London W1M 8JS
Telephone 01-486 5811 ext. 6312

Due for return on:

| | | |
|---|---|---|
| 11. 12. 89 | 06. 12. | 29. JAN 97 |
| 14. 12. 89 | 10. 01. 94 | 04. FEB 97 |
| 13. 02. '90 | 17. 10. 94 | 03. FEB 98 |
| 14. 02. 91 | 21. 11. 94 | |
| 26. 02. 92 | 15. 02. 95 | |
| 09. 03. 92 | 07. JUL 95 | |
| 25. 06. | 22. APR. 96 | |
| 21. 10. | | |
| 12. 07. 93 | | |
| 03. 11. | | |

Failure to return or renew overdue books may result in suspension of borrowing rights at all PCL libraries

20 0214027 7

# AN INTRODUCTION TO SOLID MODELING

# PRINCIPLES OF COMPUTER SCIENCE SERIES

ISSN 0888-2096

## Series Editors

**Alfred V. Aho,** Bell Telephone Laboratories, Murray Hill, New Jersey
**Jeffrey D. Ullman,** Stanford University, Stanford, California

1. *Algorithms for Graphics and Image Processing**
   **Theo Pavlidis**
2. *Algorithmic Studies in Mass Storage Systems**
   **C. K. Wong**
3. *Theory of Relational Databases**
   **Jeffrey D. Ullman**
4. *Computational Aspects of VLSI**
   **Jeffrey D. Ullman**
5. *Advanced C: Food for the Educated Palate**
   **Narain Gehani**
6. *C: An Advanced Introduction**
   **Narain Gehani**
7. *C for Personal Computers: IBM PC, AT&T PC 63000, and compatibles**
   **Narain Gahani**
8. *Principles of Computer Design**
   **Leonard R. Marino**
9. *The Theory of Database Concurrency Control**
   **Christos Papadimitriou**
10. *Computer Organization**
    **Michael Andrews**
11. *Elements of Artificial Intelligence Using LISP*
    **Steven Tanimoto**
12. *Trends in Theoretical Computer Science*
    **Egon Börger, Editor**
13. *An Introduction to Solid Modeling*
    **Martti Mäntylä**

*These previously-published books are in the *Principles of Computer Science Series* but they are not numbered within the volume itself. All future volumes in the *Principles of Computer Science Series* will be numbered.

## OTHER BOOKS OF INTEREST

*Jewels of Formal Language Theory*
**Arto Salomaa**

*Principles of Database Systems*
**Jeffrey D. Ullman**

*Fuzzy Sets, Natural Language Computations, and Risk Analysis*
**Kurt J. Schumucker**

*LSIP: An Interactive Approach*
**Stuart C. Shapiro**

0881751081
006.6 MAN
ENG

# AN INTRODUCTION TO SOLID MODELING

MARTTI MÄNTYLÄ
*HELSINKI UNIVERSITY OF TECHNOLOGY*

COMPUTER SCIENCE PRESS

Copyright © 1988 Computer Science Press, Inc.

Printed in the United States of America

All rights reserved. No part of this book may be reproduced in any form including photostat, microfilm, xerography, and not in information storage and retrieval systems, without permission in writing from the publisher, except by a reviewer who may quote brief passages in a review or as provided in the Copyright Act of 1976.

*Computer Science Press*
*1803 Research Boulevard*
*Rockville, Maryland 20850*

1 2 3 4 5 6 93 92 91 90 89 88

**Library of Congress Cataloging-in-Publication Data**

Mäntylä, Martti, 1955-
An introduction to solid modeling.

(Principles of computer science series ; 13)
Bibliography: p.
Includes index.
1. Computer graphics. 2. Solids. I. Title.
II. Series.
T385.M363 1988 006.6 87-27852
ISBN 0-88175-108-1

# Preface

The purpose of this book is to fill what I think is an important gap in the computer graphics literature. While there are many excellent introductory books on graphics available, a reader interested in representing and processing geometric information on three-dimensional solid objects—or just *solid modeling* for short—may have some difficulty in finding helpful literature.

Extensive use of computing technologies is generally acknowledged as one of the driving forces in the current "second industrial revolution" taking place in the manufacturing industries. Solid modeling technology has a central role in this progress by providing complete and accurate geometric information on the parts to be manufactured to the various design and manufacturing systems of the modern computerized industrial environment.

In addition to design and manufacturing, solid modeling has a role in a number of other applications as well. These include such diverse areas as computer animation, computer art, medical computing, and even mining (in the form of ore body modeling). Nevertheless, it seems that, to date, only a small portion of the potential uses of solid modeling have been discovered and brought to practice.

While solid modeling technology is still far from perfect, it is clear that insufficient knowledge of its possibilities and restrictions has slowed down the growth of its use. This book is intended for professionals both in industry and academia who are involved with the design, implementation, and use of computer systems based on solid modeling technologies. Those who are not actually developers of solid modeling systems can benefit from accurate knowledge on the internals of solid modelers as well, just as knowledge on operating systems is useful for people who do not write operating systems themselves.

## Content of the Book

The book consists of two parts. The first part, **Fundamentals**, introduces the reader to the background, the problems, and the major approaches of solid modeling. Various techniques now employed in industrial and academic systems are illustrated with the use of case studies of algorithms. A brief treatment of graphical models is also given.

The second part, **The Geometric WorkBench**, describes in detail the construction of a simple solid modeling system based on the boundary modeling approach, the Geometric WorkBench (GWB). Its chapters elaborate the mathematical concepts of boundary modeling, and introduce the data structures and the algorithms of the modeler, including advanced operations such as the Boolean set operations. Problems related to providing an interactive user interface to the modeler and increasing its geometric coverage and efficiency are also briefly discussed.

The book contains numerous sample programs shown in C code or C-like pseudocode. In addition, problem sections including programming assignments are provided at the end of each chapter.

The book does not attempt to cover the whole area of object modeling for computer graphics or computer-aided design. Instead, it presents one approach to the construction and use of solid models at a level of clarity and detail that should make it possible for the reader to construct his/her own modeling system. I am a believer in "learning by doing," and encourage the reader to work along the lines of this book, and to explore further.

## Acknowledgements

The first version of the book was written while I was visiting the Computer Systems Laboratory at the Stanford University and benefitted greatly from the financial help of the Rotary International Foundation, the Finnish Academy, the Finnish Cultural Foundation and the Helsinki University of Technology. Later, the work was continued at the Helsinki University of Technology with partial support from the Technology Development Centre.

Many people have provided their assistance during the preparation of the text. I should like to express my warmest thanks to Alain Fournier, Reijo Sulonen, and Markku Tamminen, who gave extensive reviews on early versions of the book. Helpful comments were received also from David Vanderschel, Martin Newell, Panu Rekola, Marja-Riitta Koivunen, and Timo Kunnas. Some figures appearing in the book were created by Lasse Holmström, Timo Laakko, and Mervi Ranta. My class on *"Data Structures and Algorithms for Geometric Problems"* at Stanford had to suffer early versions of the Chapters 4 through 8; the feedback of the students of the class

is gratefully acknowledged. The whole design of GWB that forms the basis of Part Two has greatly benefitted from the long and sometimes heated discussions I have had with Reijo Sulonen, Markku Tamminen, Tapio Takala, Lasse Holmström, and Timo Laakko. Without the initiative and help of Dennis Allison, the preparation of this book would never have been started.

Finally, I should like to thank my family for their love and patience during the long hours, days, weeks, and months this text was being written.

M. Mäntylä
Espoo, Finland
August, 1987

# Table of Contents


**Part One: FUNDAMENTALS** 1

**1 INTRODUCTION** 3
- 1.1 AIMS OF MODELING . . . 3
- 1.2 COMPUTER MODELS . . . 4
  - 1.2.1 Geometric Modeling . . . 4
  - 1.2.2 Solid Modeling . . . 5
- 1.3 PROBLEMS OF SOLID MODELING . . . 5
  - 1.3.1 Completeness . . . 5
  - 1.3.2 Integrity . . . 7
  - 1.3.3 Complexity and Geometric Coverage . . . 8
  - 1.3.4 Nature of Geometric Computation . . . 9
- 1.4 SOLID MODELING SYSTEM . . . 9

**2 GRAPHICAL MODELS** 13
- 2.1 BASIC MODELING PRIMITIVES . . . 13
  - 2.1.1 Points . . . 13
  - 2.1.2 Lines . . . 14
  - 2.1.3 Independent Points and Lines . . . 14
  - 2.1.4 Design Issues . . . 15
- 2.2 GROUPING . . . 18
  - 2.2.1 Objects . . . 18
  - 2.2.2 Graphical Symbols . . . 19
- 2.3 EXTENSIONS . . . 22
  - 2.3.1 Construction Techniques . . . 22
  - 2.3.2 Systems With Constraints . . . 23
  - 2.3.3 Parametric Design . . . 25

**3 GEOMETRICALLY COMPLETE MODELS** 29
- 3.1 PROBLEMS OF GRAPHICAL MODELS . . . 29
- 3.2 A THREE-LEVEL VIEW OF MODELING . . . 30

3.3 MATHEMATICAL MODELS OF SOLIDS . . . . . 32
3.4 POINT-SET MODELS . . . . . 32
3.4.1 Rigidity . . . . . 33
3.4.2 Regularity . . . . . 33
3.4.3 Representational Finiteness . . . . . 34
3.5 SURFACE-BASED MODELS . . . . . 35
3.5.1 2-Manifolds . . . . . 35
3.5.2 Limitations of Manifold Models . . . . . 36
3.5.3 Plane Models of 2-Manifolds . . . . . 36
3.5.4 Formal Definition of Plane Models . . . . . 38
3.5.5 Realizable Plane Models . . . . . 41
3.5.6 The Euler Characteristic . . . . . 43
3.5.7 Surfaces With Boundary . . . . . 46
3.5.8 Duality . . . . . 47
3.5.9 Wrap-Up of Plane Models . . . . . 48
3.6 REPRESENTATION SCHEMES . . . . . 49
3.6.1 Properties of Representation Schemes . . . . . 50
3.7 PRIMITIVE INSTANCING . . . . . 52
3.7.1 Concepts of Primitive Instancing . . . . . 52
3.7.2 Properties of Primitive Instancing . . . . . 54
3.8 A TAXONOMY OF SOLID MODELS . . . . . 55

**4 DECOMPOSITION MODELS 59**
4.1 EXHAUSTIVE ENUMERATION . . . . . 59
4.1.1 Construction of Exhaustive Enumerations . . . . . 60
4.1.2 Uses of Exhaustive Enumeration . . . . . 61
4.1.3 Properties of Exhaustive Enumeration . . . . . 62
4.2 SPACE SUBDIVISION SCHEMES . . . . . 62
4.2.1 The Octree Representation . . . . . 63
4.2.2 Binary Space Subdivision . . . . . 70
4.2.3 Linearized Space Subdivision . . . . . 70
4.2.4 Geometric Search by Subdivision . . . . . 72
4.3 CELL DECOMPOSITIONS . . . . . 72
4.3.1 Representation of Cell Decompositions . . . . . 72
4.3.2 Properties of Cell Decompositions . . . . . 73

**5 CONSTRUCTIVE MODELS 77**
5.1 HALF-SPACE MODELS . . . . . 77
5.1.1 Half-Spaces . . . . . 77
5.1.2 Boolean Set Operations . . . . . 78
5.1.3 Representation of Half-Space Models . . . . . 79
5.1.4 Properties of Half-Space Models . . . . . 80

5.2 CONSTRUCTIVE SOLID GEOMETRY . . . . . 81
5.2.1 Representation of CSG Models . . . . . 81
5.2.2 Algorithms for CSG Models . . . . . 84
5.2.3 Generalized CSG Primitives . . . . . 96
5.2.4 Properties of CSG Models . . . . . 97

**6 BOUNDARY MODELS 101**
6.1 BASIC CONCEPTS . . . . . 101
6.2 BOUNDARY DATA STRUCTURES . . . . . 102
6.2.1 Polygon-Based Boundary Models . . . . . 103
6.2.2 Vertex-Based Boundary Models . . . . . 104
6.2.3 Edge-Based Boundary Models . . . . . 105
6.2.4 Faces With Several Boundaries . . . . . 108
6.3 VALIDITY OF BOUNDARY MODELS . . . . . 110
6.4 DESCRIPTION OF BOUNDARY MODELS . . . . . 111
6.4.1 Conversion from CSG . . . . . 112
6.4.2 Two-and-Half-Dimensional Drawing . . . . . 113
6.4.3 Local Modification . . . . . 114
6.5 ALGORITHMS FOR BOUNDARY MODELS . . . . . 115
6.5.1 Visualization . . . . . 115
6.5.2 Integral Properties . . . . . 116
6.6 PROPERTIES OF BOUNDARY MODELS . . . . . 119

**7 HYBRID MODELERS 123**
7.1 WHY MULTIPLE REPRESENTATIONS? . . . . . 123
7.2 PROBLEMS OF HYBRID MODELERS . . . . . 124
7.2.1 Conversions . . . . . 124
7.2.2 Consistency . . . . . 125
7.2.3 Modeling Transactions . . . . . 125
7.3 HYBRID ARCHITECTURES . . . . . 126
7.4 DISTRIBUTED MODELERS . . . . . 128
7.5 OTHER HYBRID APPROACHES . . . . . 129

**Part Two: THE GEOMETRIC WORKBENCH 131**

**8 TOWARDS THE GEOMETRIC WORKBENCH 133**
8.1 DESIGN GOALS OF GWB . . . . . 133
8.2 ARCHITECTURE OF GWB . . . . . 134
8.2.1 Layers of GWB . . . . . 134
8.2.2 An Analogy: Computer Languages . . . . . 136

**9 EULER OPERATORS 139**
9.1 MANIPULATION OF PLANE MODELS . . . . . 139
9.1.1 Local Topological Operations . . . . . 140
9.1.2 Global Topological Operations . . . . . 142
9.1.3 Soundness of Plane Model Operations . . . . . 145
9.2 MANIPULATION OF BOUNDARY MODELS . . . . . 146
9.2.1 Notation and Conventions . . . . . 147
9.2.2 Skeletal Primitives: MVFS and KVFS . . . . . 148
9.2.3 Local Manipulations . . . . . 148
9.2.4 Global Manipulations: KFMRH, MFKRH . . . . . 151
9.3 AN EXAMPLE OF EULER OPERATORS . . . . . 152
9.4 PROPERTIES OF EULER OPERATORS . . . . . 154
9.4.1 Euler-Poincaré Formula Revisited . . . . . 154
9.4.2 Algebraic Properties of Euler Operators . . . . . 156
9.4.3 Descriptive Power of Euler Operators . . . . . 158

**10 HALF-EDGE DATA STRUCTURE 161**
10.1 REQUIREMENTS . . . . . 161
10.2 OVERVIEW OF THE DATA STRUCTURE . . . . . 162
10.2.1 Graph Representation . . . . . 163
10.2.2 Representation of Identification . . . . . 165
10.2.3 Empty Loops . . . . . 167
10.3 IMPLEMENTATION DETAILS . . . . . 167
10.4 ACCESSING THE DATA STRUCTURE . . . . . 167
10.4.1 Hierarchic Access . . . . . 171
10.4.2 Access over Edges . . . . . 171
10.5 NOTES . . . . . 171

**11 IMPLEMENTATION OF EULER OPERATORS 175**
11.1 OVERVIEW . . . . . 175
11.2 A STORAGE ALLOCATOR . . . . . 175
11.2.1 Linked List Procedures . . . . . 177
11.2.2 Halfedge Procedures . . . . . 179
11.3 LOW-LEVEL EULER OPERATORS . . . . . 179
11.3.1 MVFS . . . . . 179
11.3.2 LMEV . . . . . 183
11.3.3 LMEF . . . . . 183
11.3.4 LKEMR . . . . . 186
11.4 HIGH-LEVEL EULER OPERATORS . . . . . 186
11.5 C CALLING SEQUENCES . . . . . 190
11.5.1 Low-Level Euler Operators . . . . . 190
11.5.2 High-Level Euler Operators . . . . . 194

**12 BASIC MODELING ALGORITHMS** 199
12.1 MOTIVATION . . . . . 199
12.2 ARC GENERATOR . . . . . 200
12.3 SWEEPING PRIMITIVES . . . . . 200
12.3.1 Translational Sweeping . . . . . 200
12.3.2 Rotational Sweeping . . . . . 202
12.4 GLUING . . . . . 206
12.4.1 Joining of Solids . . . . . 209
12.4.2 Loop Gluing . . . . . 209
12.5 ROTATIONAL SWEEPING REVISITED . . . . . 213

**13 GEOMETRIC ALGORITHMS** 217
13.1 FACE EQUATIONS . . . . . 217
13.2 CONTAINMENT AND INTERSECTION ALGORITHMS 218
13.2.1 Vertex Equality . . . . . 218
13.2.2 Vertex-Edge Containment . . . . . 221
13.2.3 Vertex-Loop Containment . . . . . 221
13.2.4 Line Intersection . . . . . 226
13.3 INTEGRAL PROPERTIES . . . . . 226
13.3.1 Volume . . . . . 226
13.3.2 Surface Area . . . . . 226

**14 A SPLITTING ALGORITHM** 233
14.1 THE PROBLEM . . . . . 233
14.2 OVERVIEW . . . . . 235
14.2.1 Reduction Step . . . . . 235
14.2.2 Vertex Neighborhood Classification . . . . . 238
14.2.3 Computation of the Result . . . . . 239
14.3 OUTLINE OF THE ALGORITHM . . . . . 239
14.4 REDUCTION STEP . . . . . 240
14.5 VERTEX NEIGHBORHOOD CLASSIFIER . . . . . 242
14.5.1 Reclassification Rules . . . . . 242
14.5.2 Implementation of the Classifier . . . . . 245
14.6 INSERTION OF NULL EDGES . . . . . 248
14.6.1 Insertion Rules . . . . . 248
14.6.2 Insertion Procedure . . . . . 251
14.7 JOINING OF NULL EDGES . . . . . 251
14.7.1 Joining Order . . . . . 251
14.7.2 Implementation of the Joining Algorithm . . . . . 253
14.8 CLASSIFICATION OF FACES . . . . . 258
14.9 SLICING . . . . . 259

**15 BOOLEAN SET OPERATIONS 263**
15.1 INTRODUCTION . . . . . 263
15.2 STATEMENT OF THE PROBLEM . . . . . 265
15.3 BOUNDARY CLASSIFICATION . . . . . 267
15.4 THE ALGORITHM . . . . . 271
15.5 REDUCTION STEP . . . . . 271
15.6 VERTEX NEIGHBORHOOD CLASSIFIER . . . . . 277
15.6.1 Vertex-Face Classification . . . . . 277
15.6.2 Vertex-Vertex Classification . . . . . 278
15.7 JOINING OF NULL EDGES . . . . . 293
15.8 GENERATION OF THE RESULT . . . . . 294
15.9 FINAL REMARKS . . . . . 298

**16 UNDOING 301**
16.1 MOTIVATION . . . . . 301
16.2 OVERVIEW . . . . . 302
16.3 AN UNDO LOG DATA STRUCTURE . . . . . 303
16.4 UNDOABLE EULER OPERATORS . . . . . 304
16.5 OTHER UNDOABLE OPERATIONS . . . . . 307
16.6 TRANSACTION MANAGEMENT . . . . . 309
16.7 APPLICATIONS OF UNDOING . . . . . 313
16.7.1 Error Rollback . . . . . 313
16.7.2 File Operations of Solids . . . . . 315

**17 A USER INTERFACE 325**
17.1 LEVELS OF A MODELER REVISITED . . . . . 325
17.2 BATCH INTERFACE . . . . . 326
17.3 INTERACTIVE INTERFACE . . . . . 329
17.3.1 Interaction Architecture . . . . . 331
17.3.2 Modeling Primitives . . . . . 333
17.3.3 Anatomy of a Command Language . . . . . 333
17.4 IMPLEMENTATION . . . . . 336
17.4.1 Lexical Analyzer . . . . . 336
17.4.2 Parser and Command Procedures . . . . . 339
17.5 NOTES . . . . . 343

**18 EXTENSIONS 345**
18.1 EXTENSION OF GEOMETRIC COVERAGE . . . . . 345
18.1.1 Representation of Surfaces . . . . . 347
18.1.2 Generation of Surface Information . . . . . 348
18.1.3 Functionality with Surface Information . . . . . 350
18.2 EFFICIENT GEOMETRIC SEARCH . . . . . 352

18.2.1 Geometric Queries and Indices . . . . . 353
18.2.2 Hierarchic Indices . . . . . 354
18.2.3 Grid Indices . . . . . 356

**A HOMOGENEOUS COORDINATES** **365**
A.1 DEFINITION . . . . . 365
A.2 COORDINATE TRANSFORMATIONS . . . . . 365
A.2.1 Translation . . . . . 366
A.2.2 Rotation . . . . . 366
A.2.3 Scaling . . . . . 367
A.3 COMBINATION OF TRANSFORMATIONS . . . . . 367
A.4 A VIEWING OPERATION . . . . . 368
A.4.1 Specification of the Viewing Operation . . . . . 368
A.4.2 Construction of the Viewing Matrix . . . . . 370
A.5 A VECTOR AND MATRIX PACKAGE . . . . . 371

**B ELEMENTS OF POINT SET TOPOLOGY** **375**
B.1 CONTINUOUS TRANSFORMATIONS . . . . . 375
B.2 OTHER TOPOLOGICAL CONCEPTS . . . . . 376
B.3 TOPOLOGICAL SPACES . . . . . 377

**BIBLIOGRAPHY** **379**

**LIST OF PROGRAMS** **391**

**INDEX** **395**

# Part One

# FUNDAMENTALS

The first part introduces the reader to solid modeling, its role in computer-aided design and manufacturing, and major approaches to the construction of graphical and solid models.

# Chapter 1

# INTRODUCTION

## 1.1 AIMS OF MODELING

In general, a *model* is an artificially constructed object that makes the observation of another object easier. To make the obvervation possible, *physical models* of three-dimensional physical things such as buildings, ships, and cars usually share the relative dimensions and general appearance of their physical counterpart, but not the size. *Molecule models* used in chemistry share the relative arrangement of the various atoms of the molecule with respect to each other, but very few of its other properties. *Mathematical models*, widely used in various fields of science and engineering, represent some of the behavioral aspects of the phenomenon modeled in terms of numerical data and equations.

*Engineering drawings*, widely used in design and engineering, can also be viewed as models. Through a collection of two-dimensional images, aided by conventional symbolism for representing dimensions, tolerances, surface characteristics, materials, and so on, they are capable of conveying true three-dimensional information to an experienced reader.

Models are useful because often one can study certain characteristics of an object more easily based on the model than its physical counterpart. This may be so because the object does not yet exist (as often is the case for physical models), or because it is not directly observable (as is the case for molecules), or because it cannot be examined in a controlled fashion (as often is the case for mathematical models). Note, however, that physical and mathematical models are limited in the scope of their usefulness: a problem of a new type often requires a new model.

As models, engineering drawings try to avoid these problems. They are relatively *general-purpose*: a drawing can be used to extract information for

many tasks, including the production of physical and mathematical models when they are needed. From this universality it follows that they can act as a *medium of communication* between the people that participate in the design of an object. Unlike physical and mathematical models, drawings also support the iterative, "sketchy" nature of design. They can be fuzzy and inaccurate when appropriate, whereas physical and mathematical models are tools for representing or analyzing a relatively finished object.

## 1.2 COMPUTER MODELS

*Computer models* consist of data stored in computer files that can be used to perform similar tasks as the other kinds of models discussed above, with the aim of sharing their merits and avoiding their problems. In particular, we aim at general-purpose models that can support a wide variety of applications, just like engineering drawings.

### 1.2.1 Geometric Modeling

The totality of data that would be stored in a computer model depends on the scope of questions one wants to be able to answer, and *a priori* it does not seem possible to limit the amount of potentially interesting data. At first sight, this may appear to make computer models unattractive.

The key observation that motivates this book is that many problems that we try to solve through models are inherently *geometric*. For instance, the problem of calculating a shaded image of an object includes geometric problems such as the following:

1. Which parts of the object are visible to the viewer?

2. What color should be assigned to each element of the image?

Similarly, the problem of generating control sequences for a numerically controlled machine tool is basically geometric: which sequence of movements of the tool will produce the shape of the object?

If we can represent the geometric shape of the object adequately for these questions, we would be able to provide answers to many others as well. In fact, it appears that the geometry of an object is the most useful part of the bulk of potential information. Moreover, techniques for storing and processing geometric data are relatively independent of particular applications: essentially identical methods can be used to construct models of machines, ships, chemical plants, and baby food containers.

Accordingly, it makes sense to separate the data dealing with the geometric shape of an object from other, nongeometric data. In this approach,

the totality of data needed for a particular scope of problems is called the *object model*, while the purely geometric part of it constitutes a *geometric model*. A geometric model, of course, is a subset of the object model.

### 1.2.2 Solid Modeling

*Solid modeling* is a branch of geometric modeling that emphasizes the general applicability of models, and insists on creating only "complete" representations of physical solid objects, i.e. representations that are adequate for answering arbitrary geometric questions *algorithmically* (without the help of interaction with a human user).

The goal of relatively general applicability separates solid modeling from other types of geometric models which are biased towards some special purposes. *Graphical models* are intended for describing a *drawing* of an object rather than the object itself. *Shape models* represent an *image* of an object. They may be unstructured collections of image elements, or may have some internal structure to aid, say, image processing operations. *Surface models* give detailed information on a curved surface, but do not always give sufficient information for determining all geometric properties of the object bounded by the surface.

Because of our interest in general-purpose modeling, we exclude these other types of computer models from the scope of this book. Nevertheless, as we shall see, some methods developed for the construction of these models have found their way to solid modeling as well.

## 1.3 PROBLEMS OF SOLID MODELING

The requirement of general applicability demands much of the completeness and accuracy of solid models. In the following, we discuss on an intuitive level some of the problems we shall deal with in later parts of this book; in Chapter 3 we shall take a more systematic look into them.

### 1.3.1 Completeness

The simplest kinds of computer models of physical objects transfer the methods used in engineering drawings directly to a computer to create *two-dimensional graphical models*. These models consist of two-dimensional graphical objects such as lines, arcs, text, and other notation needed in the figure.

Graphical models are perfectly appropriate to make the generation of technical drawings more efficient and to enhance their quality. Unfortunately, whereas human beings (or at least engineers) are capable of inter-

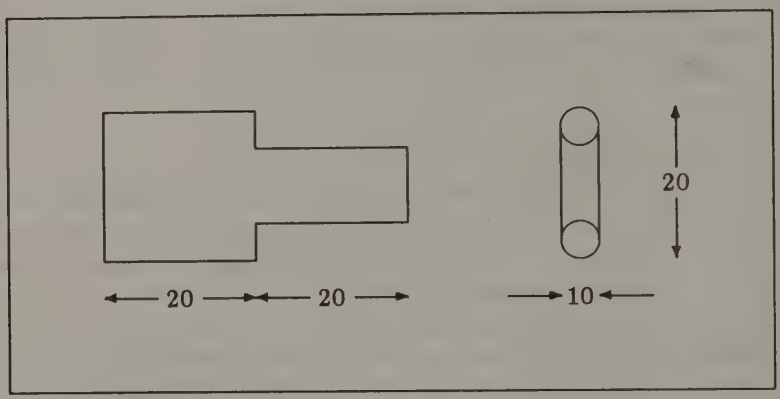



**Figure 1.1** A nonsense drawing.

preting engineering drawings properly, computers in general lag far behind them in that skill. Thus graphical models in general cannot serve as solid models.

One of the problems of graphical models is that it is perfectly possible to make drawings that represent no actual shape; see, for instance, Figure 1.1 [105]. Still worse, the problem of algorithmically inferring correct three-dimensional information from a collection of two-dimensional figures remains unsolved for practical purposes, despite some interesting results gained with simplified problems.

With the aid of some computer power, two-dimensional graphical models can be upgraded to their three-dimensional counterparts by adding the information of the third coordinate. This results in a solid representation commonly referred to as the *wire frame* model. With wire frames, it becomes possible to store only a single three-dimensional model and generate all needed two-dimensional views from it. Hence one of the problems of two-dimensional graphical models can be overcome, because it is less easy to make drawings whose views are inconsistent.

Unfortunately, even a collection of three-dimensional lines is not sufficient for representing a shape, because some collections of lines may have several interpretations in terms of solid objects. The standard example is given in Figure 1.2.

The problem of graphical models is that they do not offer *complete* geometric information of the shape of the object—information that would be sufficient for calculating answers to arbitrary geometric questions. Nev-

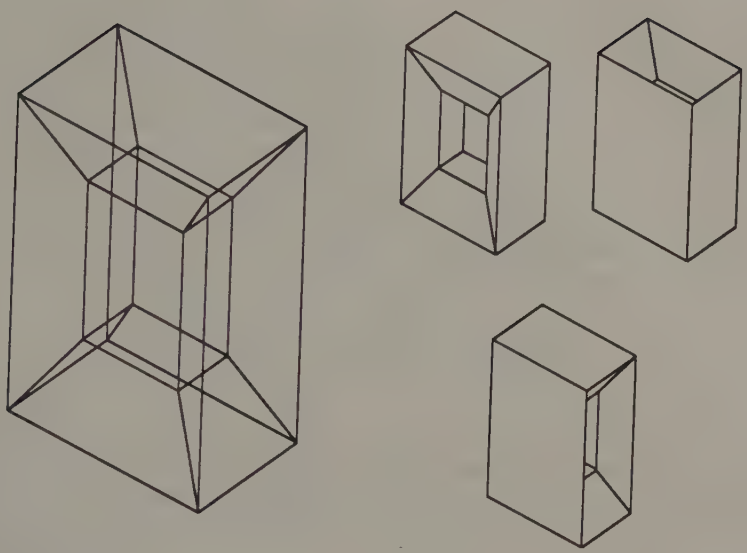


**Figure 1.2** An ambiguous graphical model.

ertheless, graphical models are perfectly all right for the task they were originally developed for, namely generation and representation of drawings. We shall study them further in Chapter 2.

### 1.3.2 Integrity

To solve the hidden line and hidden surface removal problems, the normal step taken in computer graphics methodology is to replace graphical models by *polyhedral models* that give sufficient information for determining the hidden parts of objects. These models are constructed from two-dimensional primitives, polygonal faces, rather than just lines.

Unfortunately, new problems arise. Usually, hidden line removal algorithms assume that the polygons do not intersect each other except at common edges or vertices. Obviously, a properly constructed polyhedral model of a physical shape cannot include intersecting polygons, because the surface of the object would otherwise intersect itself; hence, it is natural to consider only non-selfintersecting polyhedral models valid. But how to ensure that models satisfy criteria for correctness such as this?

The *integrity* of solid models is one of the central issues of this book. Solid models are of little value if the construction of correct models is

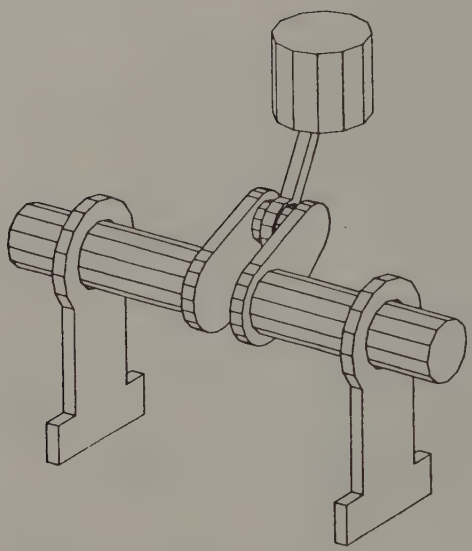


**Figure 1.3** A polyhedral model.

excessively complicated. Note that mere integrity *checking* (say, detection of illegal intersections in the hidden line removal example) is only a partial solution to the problem of integrity, because the user must figure out the corrective actions to be taken himself, and a "trial-and-error" sequence is likely to emerge.

Integrity *enforcement* would avoid this problem by making the generation of incorrect models impossible. The new problem of providing sufficient integrity checks without penalizing in the ease of use and the flexibility of the modeling system now arises, however.

### 1.3.3 Complexity and Geometric Coverage

The problem of integrity is related to another problem, the *complexity* of generating a polyhedral model. Even a relatively simple object such as the one of Figure 1.3 requires more than one hundred polygons. It is a complicated, boresome, and error-prone task to generate such information by hand.

The geometric coverage of a polyhedral model is not sufficient for tasks that require accurate modeling of curved shapes—say, automobile body design. Industries that need such models developed very early methods

1. *What does the object look like?*
2. *What is the weight, surface area, etc. of the object?*
3. *Does this object hit this other one as they move?*
4. *Is the object strong enough to carry this load?*
5. *How can the object be manufactured with certain available manufacturing processes?*

**Table 1.1** Geometric questions.

for dealing with quite complex shapes. Unfortunately, the difficulty of dealing with solid geometry in a computer rises sharply with the complexity of mathematical formulations used. Not very many solid modelers offer complete functionality for quadric surfaces and tori, and probably none for freeform surfaces.

### 1.3.4 Nature of Geometric Computation

According to our desire of general applicability, we expect solid models to be capable of providing *algorithmically* answers to typical geometric questions arising in engineering applications. Examples of such questions are listed in Table 1.1.

Observe that the result of a geometric question may be an image, a single number, or a Boolean constant (*true*, *false*). In fact, it can be another solid model that models the result of the calculation, as with the question "*what is the effect of this manufacturing process applied to this object?*" Clearly, it is important that a geometric modeler includes facilities to model not only physical objects, but also effects of *physical processes* applied to them. Of course, the modeler should be capable of repeated application of these operations to the results of previous similar operations. In other words, operations available in the modeler should constitute a *closed system* where all operations are guaranteed to maintain the correctness of the underlying models.

## 1.4 SOLID MODELING SYSTEM

One of the underlying ideas of geometric modeling is that it makes sense to separate modeling from the applications, and to seek modeling techniques

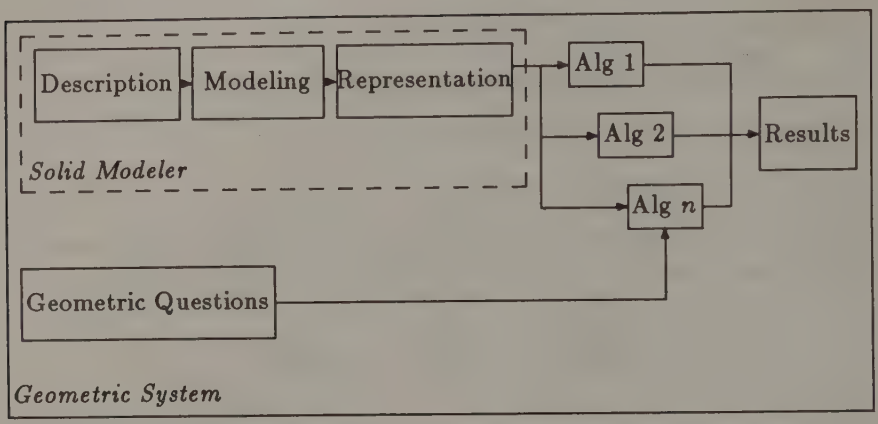



**Figure 1.4** Functional components of solid modeler.

that are relatively independent of the particular objects modeled, and of the intended use of the models. Hence, is it possible to functionally distinguish from the application proper a piece of software that deals with solid models and provides answers to geometric questions such as those of Table 1.1—the *solid modeling system* or just *solid modeler* for short.

In addition to the problems of completeness, integrity, complexity, and geometric coverage discussed in the preceding section, there are many practical points that make the construction of a solid modeler a nontrivial task. To see why this is so, let us examine the functional components of a solid modeler and its role in a "geometric system" as depicted in Figure 1.4 [95].

Initially, objects are described for the modeler in terms of a *description language* based on the *modeling concepts* available in the solid modeler. The user may enter the description simply as written text, or preferably through a *user interface* with help of graphical interaction.

Once entered, object descriptions are translated to create the actual *internal representations* stored by the modeler. The relationship between the description language and the internal representations need not be a direct one: internal representations can employ modeling concepts different to the original description. Moreover, a solid modeler may well include several description languages intended for different kinds of users and applications.

The modeler must provide interfaces for communicating with other systems. These interfaces are used to transmit information for various algorithms, or perhaps complete solid models for other design systems. The

modeler must also include facilities for storing object descriptions and other data stored by the modeler into permanent data bases.

A solid modeler must necessarily handle geometric data in different, coexisting representations. The description language, of course, forms a solid model as such. For purposes of calculating answers to geometric questions, a transformation from the external description to an internal representation is necessary. In fact, for efficiency a solid modeler should support multiple internal representations of objects; hence, it must include *conversion algorithms* that can modify data from one representation to another.

## PROBLEMS

1.1. List some other kinds of models you are familiar with. Can these models be represented in a computer?

1.2. In Mozart's opera *The Magic Flute*, prince Tamino fells in love with Pamina just by looking at her picture. To which extent is Tamino using a model in a reasonable fashion?

1.3. The extent to which the questions of Table 1.1 can really be considered "geometric" is actually debatable. List the actual information required for answering each of the questions. What information can be expected to be available in a generic geometric modeling system?

## BIBLIOGRAPHIC NOTES

The separation of various kinds of geometric models is not strict, and the identical data structures and algorithms may be applicable in several kinds of models. For instance, picture files of graphics systems may consist of exactly identical primitives as geometric models of printed circuit boards, and many algorithms are of equal interest in both cases. Here, the distinction is more a statement of the intent than of the content of the model.

The separation of "modeling" and "graphics" is nevertheless one of the fundamental ideas behind the current standard device-independent graphics software packages such as GKS [57]. The roots of this distinction can be found from a conference held in Seillac, France in 1976 [47]; see also [89].

The design and implementation of "geometric systems" in the sense of Figure 1.4 is beyond the scope of this book. For further discussion on the architectures of computer-aided design systems, see e.g. the book by Encarnação and Schlechtendahl [37] and the proceedings [36].

# Chapter 2

# GRAPHICAL MODELS

Graphical models represent geometric information in terms of points and lines. As many of the techniques for processing graphical models are of interest also to solid modeling, it is appropriate to study them more closely.

## 2.1 BASIC MODELING PRIMITIVES

The construction of a graphical model is based on a selection of *modeling primitives*, and a collection of *modeling procedures* for their instantiation and manipulation. Let us start by examining the very basic modeling primitives of graphical models: points and lines.

### 2.1.1 Points

The simplest geometric object that we must be able to represent is a location in space, a single point. We shall use a 3-dimensional right-handed Cartesian coordinate system with axes $x$, $y$, and $z$ to indicate a location in space, although other coordinate systems (such as cylindrical coordinates) can be of interest in special cases.

To represent and process points in a computer, we shall use the so-called *homogeneous coordinate representation* that adds a fourth coordinate $w$, as explained in Appendix A. Homogeneous coordinates are convenient because many operations of interest can be performed simply by multiplying the vector $[x\ y\ z\ w]$ with a 4-by-4 matrix. In particular, arbitrary sequences of *translation*, *rotation*, and *scaling* operations on $x$, $y$, and $z$ can be combined by multiplying together appropriate 4-by-4 matrices explained in the Appendix.

```
typedef struct
{
        float       vx, vy, vz, vw;
        /* other information can be added here */
} point;

ex1()
{
        point       *p;

        p = newpoint(0.0, 0.0, 0.0);
}
```

**Program 2.1** Type *point*.

To represent the point $(x\ y\ z)$, we need a data type *point*, having components *vx*, *vy*, *vz*, and *vw*. Program 2.1 gives a definition in C of this information. The procedure `newpoint` is assumed to allocate storage for a new *point* node and store the coordinate values into it. Note that points may be associated with other, problem-dependent data. In Program 2.1, we have represented only the essential geometric data explicitly.

### 2.1.2 Lines

Points alone would be quite inconvenient for modeling interesting figures. As they have just a location in space, we would not be able to create and process geometric objects having extent. The usefulness of points comes from the observation that straight line segments (that consist of infinitely many points) can be represented indirectly but completely in terms of their end points. A straightforward approach is to represent line segments in terms of two points. In this approach points can be created only as a part of a line segment. Program 2.2 gives the C definition of a type *line* that implements this, using the definitions of Program 2.1.

### 2.1.3 Independent Points and Lines

When dealing with multiple line segments, the approach reflected in Program 2.2 suffers from poor storage utilization because end points shared by several line segments are duplicated in each. Furthermore, that two line segments share an end point can only be detected by comparing the coordinate values, a slow and numerically risky operation. Another way

```
typedef struct
{
        point       p1;
        point       p2;
} line;

ex2()
{
        line        *l1;

        l1 = newline(0.0, 0.0, 0.0, 1.0, 0.0, 0.0);
}
```

**Program 2.2** Type *line*.

of structuring the information that avoids this replaces multiple copies of a point by references to a single copy. In C, a natural way to implement this is to use separate records for points and line segments, and link them together with pointers. An implementation of types *point* and *line* that follows this approach is given in Program 2.3.

### 2.1.4 Design Issues

The difference between the two representations may seem to be small. Nevertheless, it is important and illustrates the nature of design decisions that must be done during the design of geometric systems. Some problems and issues are discussed below.

#### Description of Figures

Both data representations allow the implementation of similar procedures for the description of simple line figures, such as boxes, stroke-precision characters, and symbols. However, the "flat" design of Program 2.2 makes the implementation of certain kinds of operations inconvenient.

Consider, for instance, an operation where a box (four-sided rectilinear object whose sides are parallel to coordinate axes) is created by indicating its two opposite corners. In the first approach, the natural implementation reads two locations of a pointing device, and goes on to create the box. In the second approach it is natural to let the user indicate two existing points, and create the box using their stored coordinates. Obviously, it is much easier to create boxes that "match" in the second alternative.

```
typedef struct
{
        float       vx, vy, vz, vw;
} point;
typedef struct
{
        point       *p1;
        point       *p2;
} line;

ex3()
{
        point       *pnt1, *pnt2;
        line        *l1;

        pnt1 = newpoint(0.0, 0.0, 0.0);
        pnt2 = newpoint(1.0, 1.0, 1.0);
        l1 = newline(pnt1, pnt2);
}
```

**Program 2.3** Another definition of *point* and *line*.

### Manipulation of Figures

The differences of the two representations are accentuated as for the manipulation of figures. Consider a simple figure, such as the cottage in Figure 2.1.

If we modify the endpoints of the line segments so as to "stretch" the cottage, the first representation forces us to repeat the same modification for all line segments that meet at a point. In the second approach, we only need to modify the coordinates once, and all line segments would "follow" the change. The approach of storing the points separately also enhances the viewing of models, because each point needs to be projected only once.

As the second approach provides independent modeling primitives for points and lines, it makes various more advanced construction techniques for figures possible. We shall discuss these additional construction techniques further in Section 2.3.

### Integrity

Small numerical errors can cause problems in the first approach. For instance, if line segments that seem to meet a point do not quite do so, an

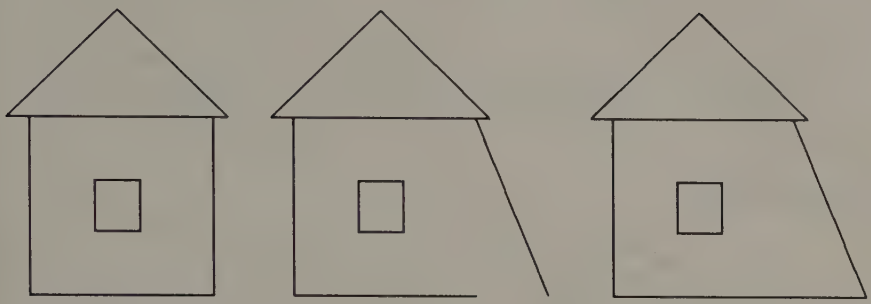


**Figure 2.1** A house.

operation that attempts to move that point could erroneously leave a line segment behind.

On the other hand, the introduction of pointers has the drawback that we have also introduced the possibility of having "dangling" pointers that refer to incorrect information. For instance, suppose that in addition to procedures **newline** and **newpoint**, the corresponding *deletion* procedures **delline** and **delpoint** are needed for the interactive manipulation of figures. Obviously, **delpoint** should be applicable only if the point to be deleted does not appear in any line segments. In the data structure of Program 2.3, testing this condition is expensive because of the necessity to scan all line segments.

A straightforward way to improve the solution is to include a new attribute *usagecount* into *point*. Initially zero, its value is incremented each time **newline** creates a line segment emanating from the point. Similarly, **delpoint** can decrement *usagecount* for the end points of the line segment it deletes. Clearly, the procedure **delpoint** can work correctly by examining **usagecount**: if it equals zero, the point does not appear in any line segments, and can safely be deleted.

Another mechanism for handling the problem is based on including an access path from each point to the all line segments adjacent to it. One way to implement this is to associate with each point a pointer to a linked list of adjacent line segments. As each line segment will appear in exactly two such lists, they can be realized by including two new pointers to each line segment node. In this approach it becomes possible to implement a more powerful point deletion operation that removes a point and all its adjacent lines. We leave the details of this and the specification of the resulting modeling procedures as an exercise to the reader.

```
typedef struct
{
        point       *allpoints;      /* list of all points */
        line        *alllines;       /* list of all lines */
} object;

typedef struct
{
        float       vx, vy, vz, vw;
        point       *nextpoint;      /* next point of the list */
} point;

typedef struct
{
        point       *p1;
        point       *p2;
        line        *nextline;       /* next line of the list */
} line;
```

**Program 2.4** Type *object*.

We choose the second representation for the purposes of this chapter; see [15] for additional discussion of the pros and cons of the two approaches.

## 2.2 GROUPING

The concepts of Program 2.3 give us the basic tools for creating graphical models of objects. In order to employ our models, we still need the capability of collecting logically interrelated points and line segments together into what might be called *groups*. This would allow our algorithms to process groups as a whole.

### 2.2.1 Objects

A new modeling primitive, *object* is needed to accomplish the grouping of related line segments and points. Objects consist of a set **allpoints** of points, and a set **alllines** of line segments. One way to implement this is to gather points and line segments belonging to the same set into linked lists, and let **allpoints** and **alllines** point to their heads. Definitions of *point* and *line* must then be modified to include a pointer to the next element in the list. These changes are implemented in Program 2.4.

Modeling procedures must be modified accordingly. One possibility is to include a pointer to the parent object as the first argument of procedures `newpoint` and `newline`, and let them add the new *point* or *line* to the correct list. Program 2.5 gives an example of the modeling procedures after these modifications.

### 2.2.2 Graphical Symbols

The grouping facility presented above is the simplest possible. A more interesting grouping mechanism allows the definition of *graphical symbols* that can be constructed through an *instantiation procedure* that assigns position and orientation information for a new *instance* of an object. In this extension, objects may consist of symbols, in addition to points and line segments.

A natural solution to this assigns a 4-by-4 transformation matrix that describes the position and orientation of each object instance relative to its parent object. (See Appendix A for information on 4-by-4 transformations.) Program 2.6 illustrates some possibilities offered by such a mechanism.

In the program, procedure `instanceof` creates a new instance of the object given as the first argument, transformed with the matrix given as the second argument. Such an instance can be added to an object with procedure `newinstance` analogously to procedures `newpoint` and `newline` of Program 2.4. Observe how the procedure can create a staircase pyramid from three instances of a single box.

The symbol data structure forms a directed acyclic graph which must be traversed when an object including symbols is "evaluated." In this process, the standard technique is to maintain a transformation stack during the traversal. Initially, an identity transformation is pushed onto the stack. When advancing to a child symbol, the current top of the stack is multiplied by the transformation of the child, and pushed onto the stack. This gives the correct transformation for positioning the child properly with respect to its parent. When the processing of a child is finished (perhaps after recursive evaluations of its children), the top of the transformation stack can be removed.

Symbol schemes are useful when pictures are formed from many instances of a small number of basic objects; consider, for instance, circuit diagrams. Correspondingly, techniques for symbol manipulation belong to the core of computer graphics methodology. We shall not delve deeper into graphical symbols; for a good exposition on the subject, the reader is referred to [39].

```
object  *newbox(dx, dy, dz)
float   dx, dy, dz;
{
        object      *box;
        point       *p1, *p2, *p3, *p4, *p5, *p6, *p7, *p8;

        /* make box an empty object */
        box = newobject();

        /* create points is the box */
        p1 = newpoint(box, 0.0, 0.0, 0.0);
        p2 = newpoint(box, dx, 0.0, 0.0);
        p3 = newpoint(box, dx, dy, 0.0);
        p4 = newpoint(box, 0.0, dy, 0.0);
        p5 = newpoint(box, 0.0, 0.0, dz);
        p6 = newpoint(box, dx, 0.0, dz);
        p7 = newpoint(box, dx, dy, dz);
        p8 = newpoint(box, 0.0, dy, dz);

        /* create lines of the box */
        newline(box, p1, p2);
        newline(box, p2, p3);
        newline(box, p3, p4);
        newline(box, p4, p1);
        newline(box, p5, p6);
        newline(box, p6, p7);
        newline(box, p7, p8);
        newline(box, p8, p5);
        newline(box, p1, p5);
        newline(box, p2, p6);
        newline(box, p3, p7);
        newline(box, p4, p8);

        return(box);
}
```

**Program 2.5** Definition of a box.

```
typedef float matrix[4][4];

object  *pyramid()
{
        object      *b, *pyr1, *pyr2, *pyr3;
        instance    *s1, *s2, *s3, *s4;
        matrix      m1, m2;

        b = newbox(1.0, 1.0, 0.25);

        ident(m1);
        trans(m1, -0.5, -0.5, 0.0);
        ident(m2);
        scale(m2, 1.2, 1.2, 1.0);
        trans(m2, 0.0, 0.0, -0.25);

        pyr1 = newobject();
        s1 = instanceof(b, m1);
        newinstance(pyr1, s1);

        pyr2 = newobject();
        s2 = instanceof(pyr1, m2);
        newinstance(pyr2, s1);
        newinstance(pyr2, s2);

        pyr3 = newobject();
        s3 = instanceof(pyr2, m2);
        newinstance(pyr3, s1);
        newinstance(pyr3, s3);
}
```

**Program 2.6** Pyramid.

## 2.3 EXTENSIONS

The very basic functionality developed so far is adequate only for illustrating the problems of, and techniques for graphical models. A practical system should include various extensions both for increased functionality and ease of use. Some of these extensions are discussed below.

### 2.3.1 Construction Techniques

For the time being, we can construct graphical models from two modeling primitives, *point* and *line* that can be grouped to form *objects*. Our modeling system is capable of

1. creating new *points* at desired locations,

2. creating *lines* between two existing *points*, and

3. scanning through *lines* and *points* of an *object*.

Clearly, by just these constructs we can model all possible line figures, so as for expressive power, we are done! However, for practical purposes many other construction methods for points and line segments would be useful as well. These additional constructions may include the following:

1. In plane, construct a line segment parallel to a given line, of given length, and starting from a given point. Length is measured along the direction of the argument line.

2. Construct a line segment perpendicular to a given line, of given length, and starting from a given point. The direction of the line segment is chosen to be 90 degrees counterclockwise from the direction of the argument line.

3. Construct a point as the intersection of two lines.

4. Construct a point as the mirrored image of a point with respect to a line.

Note that some of these operations may create many points and line segments in one shot.

Commercial drafting systems include all these and many other construction methods as well, including functionality for creating circles and arcs in terms of existing points, lines, and arcs. See [105] for a compact account of various construction methods and their implementations.

### 2.3.2 Systems With Constraints

Suppose the additional construction operations described in Section 2.3.1 above are implemented on top of the data structure of Program 2.4 (perhaps extended as suggested in Section 2.1.4). Let us consider the implementation of a new procedure `modifypoint(p, newx, newy, newz)` that assigns new coordinates to point.

Suppose further that a line segment $l_1$ starting from point $p$ and of given length is created to be parallel to a line $l_0$. What happens if we update the coordinates of $p$? After the operation, $l_1$ (and all other line segments emanating from $p$) has been modified, and in all likelihood $l_1$ is not anymore parallel to $l_0$.

This effect can be perfectly acceptable in some applications. Sometimes, however, we would like to see the line segment $l_1$ move with $p$ while remaining parallel to $l_0$. Hence, the other end point of $l_1$ must move as well, and any line segments potentially emanating from it move according to how they were constructed. This facility is useful, say, when dealing with *rigid* objects whose shape is invariant under movement.

#### Representations for Constraints

To implement such a scheme we must be able to associate constraints with each line segment and point. For simplicity, let us discuss how we could store the constraints in the case of just two construction methods of line segments, namely

1. construct a line segment between two points
2. construct a line segment parallel to another line, starting at a point, and of given length.

Observe that the latter construction creates both a line segment and a point. It can be implemented as first constructing the other end point of the line segment as determined by the arguments, and then constructing the resulting line segment between the two end points.

Constrained graphical models generally break the representation of each entity into two parts. Data that are independent of the construction method by which the entity was created are referred to as the *canonical* representation of the entity, whereas the remaining data forms the *reference* representation of the entity.

Let us use the representation of Program 2.4 as the canonical representation of our line segments and points. To include the reference information into line segments and points, we add a new attribute *refs* for storing a pointer to a node that records the construction method used for the

```
typedef struct
{
        int         code;           /* = 0 */
        float       x, y, z;
} point_from_coordinates;

typedef struct
{
        int         code;           /* = 1 */
        line        *l;
        point       *p;
        float       *l;
} point_at_distance_from_point_along_line;
```

**Program 2.7** Points with constraints.

line segment or the point. These nodes include a code that identifies the particular construction method they represent, and a list of arguments of the construction procedure. The arguments may be pointers to geometric entities or numerical values.

As our line segments are always bounded by two points, it turns out that only points need to be constrained. In our example, we need the two constraint nodes depicted in Program 2.7 for the two different kinds of points: (1) those that simply are created from a triple of coordinates (like in `newpoint`), and (2) those that are created in the construction above as the other end point of the parallel line segment.

### Regeneration

Let us now return to the implementation of `modifypoint` with the constrained data structures of Program 2.7. Obviously, the operation is meaningful only if the point is of type (1) above (why?). After checking this, the algorithm should *regenerate* all line segments and points whose canonical representation depends on the modified point either directly or through a sequence of any constructions.

Observe that the construction information defines a partial order of line segments and points[1]. Let us write $a < b$ to denote that $a$ is defined in terms of $b$. To perform the regeneration operation correctly, we must process all objects $e < p$, the modified point. Moreover, we must assure

[1]This is so because the constrained model was created by a linear sequence of modeling operations in the first place.

that if $a < b$, $b$ is regenerated before $a$.

One way to accomplish this is to link all points and line segments into yet another linked list in the order they were created. The regeneration algorithm can then consider points and line segments in the order they appear in the list. However, this solution has the drawback that lots of unnecessary work is performed if only a few entities must be updated. At the cost of additional stored information, a more efficient approach is to perform a search in the graph defined by constrain relationships while taking the relation $<$ into account.

#### Unbounded Graphical Entities

Some graphical modelers make a distinction of *bounded* graphical entities (such as our line segments) and *unbounded* graphical entities. In these systems lines are the truly unbounded lines of mathematics, whereas a separate entity is used for representing bounded line segments.

While increasing the complexity of the modeler, this separation allows the inclusion of additional construction methods and simplifies the specification of many constructions. For instance, a parallel line could be constructed just by giving a distance from the argument line, whereas a starting point and a length would be additionally required for the creation of a bounded line segment.

Observe that if unbounded graphical entities are used, both lines and points would be constrained. For instance, a parallel line constructed as above would depend on the argument line. Points created on the line would depend on it, and line segments adjacent to those points on them.

### 2.3.3 Parametric Design

Constrained graphical data structures bring us to data structures useful for a *drafting system*. A drafting system allows the user to interactively construct figures from simple geometric entities. A full-scale system would include all facilities discussed in this chapter, including grouping (perhaps in several ways), symbols, many construction methods, and constrained data structures. It would probably include additional modeling primitives, such as arcs and character strings. It would also include functional components not discussed here at all, such as a graphical user interface and mechanisms for storing models into secondary storage.

One interesting characteristic of some drafting systems is the capability of supporting *parametric design*. It can be understood as a generalization of constrained data structures. Consider again the example of Program 2.7. Going a step further, let us replace the number *length* by a pointer to a

new entity *number*. This gives us the capability of storing *all* dimensions of an object as *numbers*, and referring to them in other constructions.

In fact, we can include construction methods for numbers in our modeling system. These constructions are likely to include simple mathematical expressions and geometric measurements such as distance. Like other entities, each number has a canonical format that simply gives its value, and a constrained format that represents the expression or measurement from which the number was generated. Extending the regeneration algorithm to numbers gives the capability of modifying the (few) numbers that form the roots of the whole construction, and hence generating new versions of the whole design easily.

## PROBLEMS

2.1. Write a C definition of points and lines as suggested in Section 2.1.4. That is, associate with each point the collection of lines it is adjacent to.

2.2. Write modified versions of procedures `newline` and `newpoint` that use the modified data structures of Problem 1.

2.3. Specify and write procedures `delline` and `delpoint` based on the data structures of Problem 1.

2.4. Based on the data structures of Program 2.5, write procedures `newobject`, `instanceof`, and `newinstance` discussed in Section 2.2.

2.5. Based on the data structures of Program 2.7, specify and write the procedures needed for drawings with constraints.

## BIBLIOGRAPHIC NOTES

For this book, graphical models are only of secondary interest. A reader interested on them should therefore consult additional literature.

Graphical models are the major ones currently in use in Computer-Aided Design and Drafting systems. Consequently, books on CADD [15,11,43] contain material on graphical models much beyond this chapter.

The texts of Faux and Pratt [38] and Mortenson [85] contain (in addition to a lot of other material) good overviews of curve and surface presentations. The book by Woodwark [131] gives a compact account on not only two-dimensional techniques, but also on many aspects on three-dimensional models as well. The books of Chasen [24], Rogers [103,102], and Bowyer and Woodwark [16] contain readily useful techniques for various geometric constructions (primarirly) for simple geometric objects. Commercially available CADD systems are the subject of [32].

Parametric design techniques are in use not only in drafting systems but also in the user interfaces of solid modelers. For instance, the user interface of MEDUSA [87] has a parametric design facility that includes also circular arcs and is capable of storing relationships involving tangency constraints. As a more advanced example, Lin *et al.* [71] give an algorithm that can check whether a set of such constraints fully determines a two-dimensional figure consisting of line segments and arcs. The algorithm can also flag overconstrained figures.

# Chapter 3

# GEOMETRICALLY COMPLETE MODELS

Graphical models do not satisfy our requirements for completeness and general applicability. We now start the search for other solutions by characterizing more closely the desired properties of solid models.

## 3.1 PROBLEMS OF GRAPHICAL MODELS

Chapter 2 gave us an intuitive understanding on how to construct graphical models from lines and points. While there are many practically important aspects that we have not touched at all (for instance the representation of graphical models in secondary storage, or their generalization to represent and process curves), we can now assess the representational issues of graphical models.

So, which of the questions of Table 1.1 on page 9 can be answered based on the information available in graphical models?

It is easy to generate simple graphical output based on the information in a graphical model (as the term implies). Unfortunately, points and lines alone do not, in general, convey sufficient information to create more sophisticated output involving removal of hidden lines or shaded surfaces. The least that is needed is a polyhedral model. (Of course, for a mere wire frame consisting of just points and lines, surface properties such as visibility and shading are not even defined.)

No fully automatic conversion from a graphical model to a polyhedral model can exist, as the ambiguous object of Figure 1.2 on page 7 shows. Remarkably, an algorithm capable of enumerating *all* polyhedral models

corresponding to a given graphical model has been reported by Markowsky and Wesley [80]. But human guidance is still needed to select the correct one.

For restricted classes of objects, even an automatic conversion is possible. For instance, Hanrahan [50] reports an algorithm that can construct a polyhedral model from a wire frame whose points and lines form a planar graph, i.e., a wire frame which is known to correspond with a polyhedron with no "holes."

For some problems graphical models are adequate. If we need a collision testing algorithm for finding free paths for a moving object in a scene of static objects with lots of free space around them, we may find that collision testing between the convex hulls of the objects is adequate. Then graphical models are perfectly all right, because their information is sufficient for convex hull calculation. However, if the collision testing problem involves the search of narrow free paths inside a partially finished assembly, convex hulls will always intersect and provide little help.

With graphical models, we face thus the problems of incompleteness and potential ambiguity. If we are willing to seriously limit the scope of problems we want to deal with automatically, or if we are willing to limit the class of objects to be modeled, these problems can be avoided. But such limitations are in contradiction with the aim for general-purpose models set forth in Chapter 1, and motivate our search for other, geometrically more rigorous models.

## 3.2 A THREE-LEVEL VIEW OF MODELING

In the preceeding, we used terms such as "completeness" and "ambiguity" to speak of the merits and the problems of graphical models without bothering too much about their meaning. Some examples of the previous section seem to blur the intuitive image we have of these terms, however. Are graphical models "complete" as far as convex hulls are concerned? Are "ambiguities" resolved if we wisely stick to sufficiently simple objects?

This confusion can be avoided if we adopt a more rigorous view of modeling [95]. This view is based on distuingishing between three separate levels of modeling (see Figure 3.1):

1. *Physical objects:* By means of models, our aim is to speak and argue about some real things of our three-dimensional real world. Unfortunately, assuming a Platonician view, we cannot even perceive a real-world object in its full complexity and sub-microscopic detail, much less represent all aspects of it in a computer.

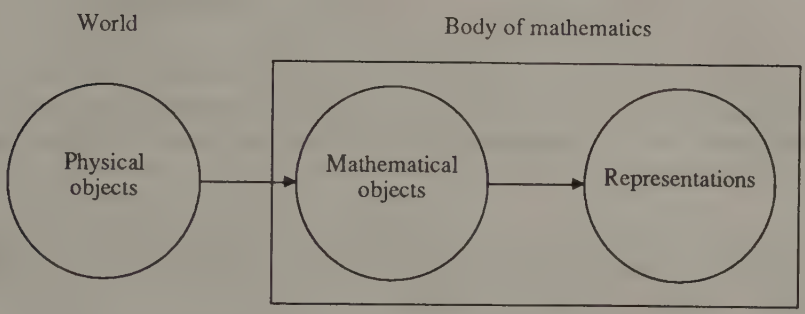



**Figure 3.1** A three-level view of modeling.

2. *Mathematical objects:* In order to have any hope of modeling objects in a computer, we must therefore adopt a suitable idealization of the real three-dimensional physical objects we are ultimately interested in. These idealized objects should have an intuitively clear connection with the real world, while being so simple that we can assign computerized representations to them.

   To achieve this, we shall characterize rigorously a class of mathematical objects (the *modeling space*) we would like to speak about. As we shall see later, such a characterization can be achieved by using basic concepts from the mathematical theories of point-set and algebraic topology.

3. *Representations:* After establishing rigorously the class of mathematical objects we are interested in, the final step of the modeling activity is to assign to the mathematical object a representation which is suitable for computer manipulation. As we shall see later, the three-level view of modeling allows us to characterize the desirable properties of solid modeling methods rigorously by analyzing the relationships of mathematical objects and their computer representations.

According to the three-level view, we shall not attempt to model actual physical properties of things. Instead, we shall try to represent some idealized objects in a manner that allows us to analyze their idealized properties, and hope that the results gained in this fashion permit us to gain information on the actual real objects hiding in the background.[1]

[1] Of course, sometimes the aim of our modeling is *not* related to physical things, but precisely to idealized things. For instance, the specification of a mechanical part defines an idealized nominal object (or a class of mechanically equivalent objects). In this case,

## 3.3 MATHEMATICAL MODELS OF SOLIDS

Let us now concentrate on sharpening our intuitive view of "solidity," i.e., on the proper definition of a class of mathematical objects that forms the subject matter of solid modeling, while leaving the exact nature of the particular geometric problems we want to solve with models aside (as for now).

It turns out that we can arrive at a characterization based on two approaches, one stressing the "three-dimensional solidity" of things, and the other concentrating on the fact that we are primarily interested on the bounding surfaces of things. These points of view lead us to using the languages of point-set topology and algebraic topology, respectively.

The next sections are aimed at introducing a rigorous characterization of the objects we shall be interested in. By necessity, they are somewhat technical; some of the required terminology is introduced in Appendix B.

## 3.4 POINT-SET MODELS

Starting from the very basics, we shall consider the three-dimensional Euclidean space $E^3$ a suitable idealization of the real space our real objects lie in. Consequently, the most general mathematical abstraction of a real solid object is a subset of $E^3$, i.e., a set of points of $E^3$.

The advantage of the point-set idealization of real objects is that we can use concepts of *point-set topology* to characterize rigorously the desired properties of three-dimensional objects. Because we shall be interested usually only in finite objects, we can start from the following working definition:

**Definition 3.1** *A* solid *is a bounded, closed subset of* $E^3$.

The restriction to closed sets (as an alternative of open sets) is a matter of convention only.

While Definition 3.1 certainly captures certain aspects of our notion of a solid, the class of objects permitted by it is far too large to make it really useful for our purposes. It turns out that to be considered "solid," a bounded, closed point set must satisfy a collection of additional requirements. The next sections will elaborate these.

---

it is the real world (the manufacturing engineer) who must imitate the idealized world, and not vice versa.

### 3.4.1 Rigidity

It is natural to expect that a solid object should remain the same if it is moved from one location to another. This can be expressed more rigorously by requesting that solids must remain invariant under *rigid transformations*, i.e., translations and rotations. The following definition captures the essence of this characterization:

**Definition 3.2** *A* rigid object *is an equivalence class of point sets of* $E^3$ *spanned by the following equivalence relation* $\circ$*: Let* $A$*,* $B$ *be subsets of* $E^3$*. Then* $A \circ B$ *holds iff* $A$ *can be mapped to* $B$ *with a rigid transformation.*

We might now add rigidity as one additional requirement to Definition 3.1. However, the usefulness of rigidity as defined above is somewhat diminished by the fact that most solid representations do not allow an efficient test for whether the relation $\circ$ is true for two models.

### 3.4.2 Regularity

Whether rigid or not, many closed and bounded point sets do not satisfy our notion of solidity. For instance, a set consisting of distinct points or lines is not solid.

We expect that a solid is "all material"; it is not possible that a single point, or a line, or even a two-dimensional area would be missing from a solid. Likewise, a "solid" point set must not contain "isolated points," "isolated lines," or "isolated faces" that do not have any material around them.

Again, these informal requirements can be compactly captured in the point-set topology language:

**Definition 3.3** *The* regularization *of a point set* $A$*,* $r(A)$*, is defined by*

$$r(A) = c(i(A))$$

*where* $c(A)$ *and* $i(A)$ *denote the closure and interior of* $A$*. Sets that satisfy* $r(A) \equiv A$ *are said to be* regular.

Informally speaking, the regularization tears off all "isolated" parts of a point set, covers it completely with a tight "skin," and fills the result up with material (see Figure 3.2).

Regularity is so widely used as a characterization of reasonable solids that the following definition (by Requicha [95]) is appropriate:

**Definition 3.4** *A bounded regular set is termed an* $r$-set.

Observe that regular sets need not be connected; for instance, the rigid combination of two regular sets is itself a regular set.

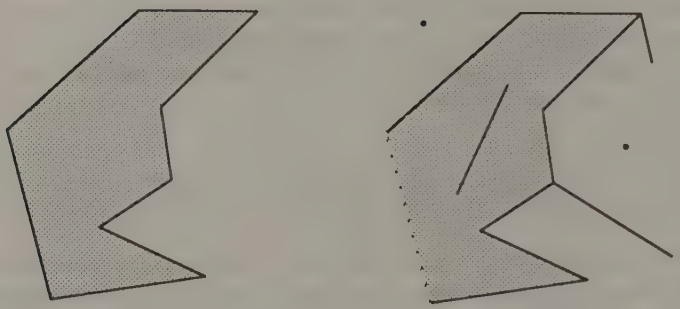


**Figure 3.2** A regular and a nonregular set.

### 3.4.3 Representational Finiteness

We have no direct means for representing continuous point sets in a computer. To model objects with dimensionality (such as a straight line), we must always employ indirect methods (such as representing the line in terms of its endpoints). Even if it is physically naive, we must therefore assume that solids have some indirect finite representation.

Usually we must require that the surface of a solid is "smooth"[2] and can adequately be modeled in terms of reasonably simple mathematical functions. There are certain subtleties involved in the "reasonably simple" of above. For instance, the set of Figure 3.3 [94] is defined by the expression

$$x, y, z : \; x_0 \leq x \leq x_1, \; y = \sin(1/x) + h, \; z_0 \leq z \leq z_1$$

that includes only (seemingly) simple functions. Yet the set does not satisfy the ordinary notion of a solid if 0 is contained in the range $[x_0 x_1]$.

To get rid of such anomalous situations, we should restrict ourselves to objects whose surfaces are algebraic (or analytic at least). That is, the surfaces must be representable by means of (finite) polynomials of $x$, $y$, and $z$.

[2]The so-called *fractal surfaces* [72,41] form a remarkable exception in this respect: they are nowhere smooth but still have an indirect, recursive representation.

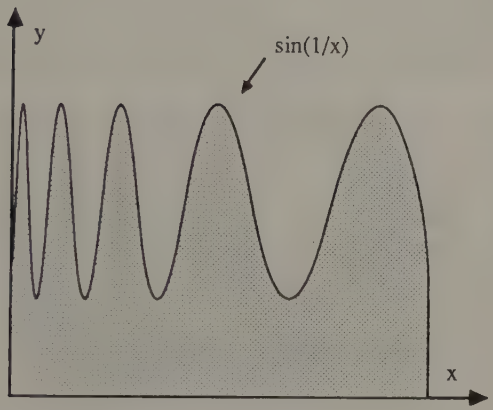



**Figure 3.3** A nonphysical solid.

## 3.5 SURFACE-BASED MODELS

As seen in the previous section, we had to refer to properties of the bounding surfaces of a solid to avoid certain anomalies not captured by the point-set view. Indeed, it is perfectly possible to characterize modeling spaces just based on properties of surfaces.

The surface-based characterization of solids looks at the boundary of a solid object. The boundary is considered to consist of a collection of "faces" which are "glued" together so that they form a complete, closing "skin" around the object. In order to be able to rigorously speak about this informal procedure, and state explicitly under which conditions we get an object satisfying our notion of "solidity," we shall use concepts developed for another branch of topology, the so-called *algebraic topology* [52,81]. Again, the following development is somewhat technical, and some concepts introduced here will be used only in Part Two of this text.

### 3.5.1 2-Manifolds

Intuitively, a surface may be regarded as a subset of $E^3$ which is essentially "two-dimensional": every point of the surface (except points on the edge of an "open" surface patch) is surrounded by a "two-dimensional" region of points belonging to the surface.

The inherent two-dimensionality of a surface means that we can study its properties through a two-dimensional model. First, we shall define a

more abstract (and precise) notion of a "surface," and then construct a simple two-dimensional model for it in terms of a special topology defined on the two-dimensional Euclidean space $E^2$.

The more abstract counterpart of a "closed surface" is defined as follows:

**Definition 3.5** *A* 2-manifold $M$ *is a topological space where every point has a neighborhood topologically equivalent to an open disk of* $E^2$.

Intuitively, a bug living on $M$ sees a continuous simple region of the surface all around itself. This intuitive view applies perfectly to ourselves and the planet Earth; indeed, the surface of a sphere is a 2-manifold.

### 3.5.2 Limitations of Manifold Models

Let us now fix our attention to an $r$-set $A$. If a 2-manifold $M$ and the boundary of $A$, $b(A)$, are topologically equivalent, we say that $A$ is a *realization* of $M$ in $E^3$. As we shall see, the class of 2-manifolds that have at least one realization can be characterized precisely; we shall call such 2-manifolds *realizable*.

Unfortunately, not all $r$-sets are realizations of some 2-manifold. Figure 3.4 depicts some such sets. The problem with all these objects is that the surface "touches" itself in a point or at a curve segment. The neighborhoods of such points are not simple disks, as required by Definition 3.5. For instance, the neighborhood of the problem point in Figure 3.4(a) consists of two disks.

Hence, there is an inherent theoretical mismatch between $r$-set models and manifold models. For practical purposes, objects such as those of Figure 3.4 can be represented indirectly as 2-manifolds by "ignoring" the exceptional points and line segments; hence the object of Figure 3.4(a) would be represented as the rigid combination of two components that just happen to touch each other at a point.

### 3.5.3 Plane Models of 2-Manifolds

To establish sufficient conditions for the realizability of a 2-manifold, we need a mechanism that can represent all 2-manifolds in a way that allows us to reason about their properties. For this purpose, we use the *plane models*.

To illustrate the basic idea of the representation, let us consider first a simple example. Figure 3.5(a) represents a four-sided polygon, while (b) represents the surface (cylinder) obtained by gluing the two edges labeled with $\alpha$ in (a) together.

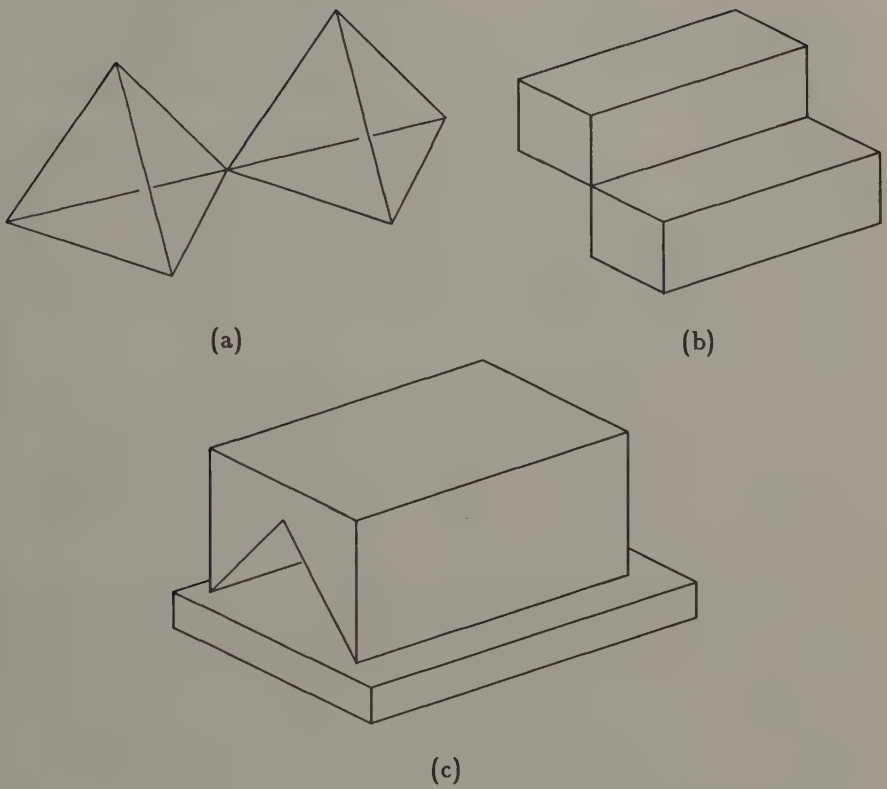



**Figure 3.4** Solids with nonmanifold surfaces.

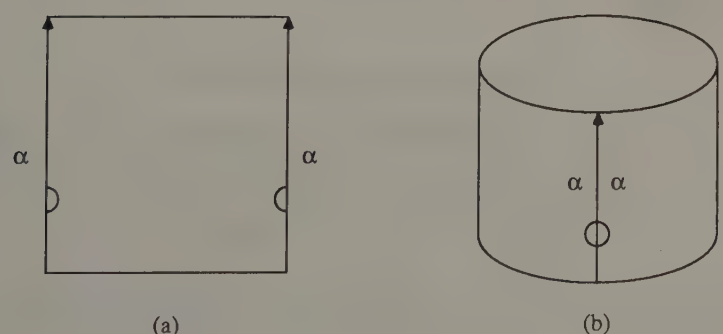



**Figure 3.5** A cylinder surface.

The basic idea of what follows is to employ a (labeled) plane figure (such as the rectangle (a)) in order to reason about the properties of a surface (the cylinder). This is done by defining a *special topology* on the plane figure that pretends that the labeled edges have already been glued together as in (b).

As noted in Appendix B, in the *natural topology* of $E^2$, all neighborhoods of points are open disks of some radius $r$ around the point. In the special topology we shall use, all neighborhoods except the neighborhoods of points on labeled edges are the same as those of the natural topology. However, the neighborhoods of those points are considered to consist of the union of two half-disks around symmetrical points on the edges (see Figure 3.5(a)).

By this special topology, symmetrical points on edges have the *same* neighborhoods, namely the disk neighborhoods they would have if the labeled edges were glued together. Such points are said to be (*topologically*) *identified*, and are treated as a single point. Observe that now all points of the cylinder, except those on the unlabeled edges of (a), have neighborhoods topologically equivalent to a disk of $E^2$. Because of these exceptional points the cylinder is called a *surface with boundary*.

Additional plane models are given in Figure 3.6. The plane model of a sphere is obtained simply by identifying the corresponding points on the two labeled edges of (a); the (American) football is an almost perfect practical example of this procedure. The natural continuation of Figure 3.5 is the model of a torus (b). It is created by identifying also the top and bottom edges of the cylinder model in a pairwise manner. (Think of bending a cylinder until its ends meet.) Observe that now all point neighborhoods are full disks, and these surfaces are *closed*.

Case (c) gives another variation of Figure 3.5: here the orientation of the identified edges have been reversed to yield the model of the well-known *Möbius strip*[3].

### 3.5.4 Formal Definition of Plane Models

In order to deal with more complicated plane models, we need to formalize the intuitive definition given above. In the following, we give a formal definition of plane models following the exposition of [52].

We start by defining the *identification* of edges:

**Definition 3.6** *Let $P$ be a set of polygons, and let $a_1, a_2, \ldots$ be a collection of edges of these polygons. These edges are termed* identified *when a new topology is defined on $P$ as follows:*

[3]The use of plane models was invented by Möbius.

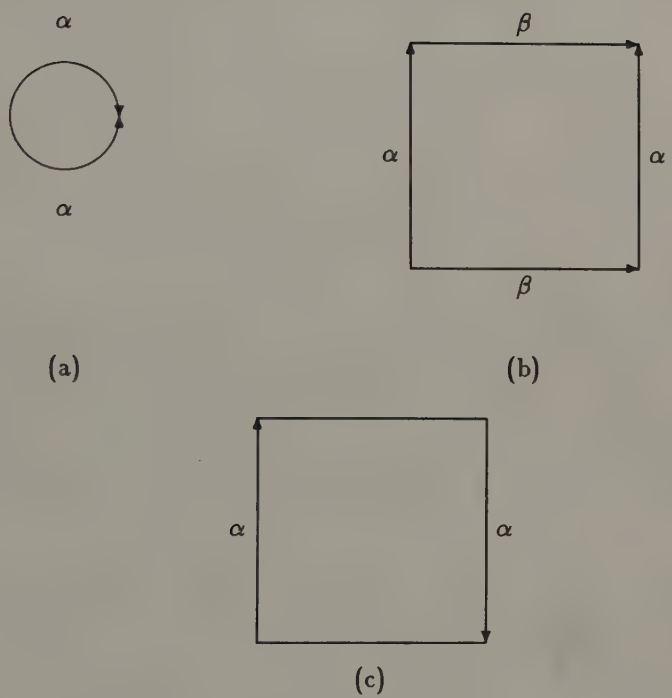



**Figure 3.6** Other surfaces: sphere (a), torus (b), and Möbius strip (c).

1. *Each edge is assigned an orientation from one endpoint to the other, and placed in topological correspondence with the unit interval in such a way that the initial points of all edges correspond to 0 and final points correspond to 1.*
2. *The points on the edges $a_1, a_2, \ldots$ that all correspond to the same value from the unit interval are treated as a single point.*
3. *The neighborhoods of the new topology on $P$ are the disks entirely contained in a single polygon plus the unions of half-disks whose diameters are matching intervals around corresponding points on the edges $a_1, a_2, \ldots$.*

In other words, in the new topology the identified edges $a_1, a_2, \ldots$ are treated as a single edge.

In the preceding examples, identification of edges was represented by labeling identified edges with the same label; arrows were used to indicate the orientation of the edges.

Next we need to define the identification of vertices:

**Definition 3.7** *Let $P$ be a set of polygons, and let $p_1, p_2, \ldots$ be a collection of vertices from these polygons. These vertices are said to be* identified *when a new topology is defined on $P$ in which this collection of vertices is treated as a single point and the neighborhoods are defined to be disks completely contained in a single polygon plus the unions of portions of disks around each of the points $p_1, p_2, \ldots$. In case any of the edges meeting at one of these vertices is also identified, the sectors forming a neighborhood at $p_1, p_2, \ldots$ must contain matching intervals from these edges.*

If a collection of vertices $p_1, p_2, \ldots$ are identified, we shall say the respective polygons $r_1, r_2, \ldots$ are identified at that collection.

The plane models can now be defined more rigorously:

**Definition 3.8** *A* plane model *is a planar directed graph $\{N, A, R\}$ with a finite number of* vertices $N = \{n_1, n_2, \ldots\}$, edges $A = \{a_1, a_2, \ldots\}$, *and* polygons $R = \{r_1, r_2, \ldots\}$ *bounded by edges and vertices. Each polygon of the graph has a certain orientation around its edges and vertices. Polygons, edges, and vertices of the graph are labeled; if a collection of edges or vertices has the same label, they are considered to be identified according to definitions 3.6 and 3.7 above.*

Figure 3.7(a) illustrates the definition above. In the figure, edges and vertices with corresponding labels are identified. Hence, for instance, the neighborhoods of each point on the edge with the label $e_1$ consist of two half-disks, and the neighborhood of the vertex $v_1$ consists of four disk sectors.

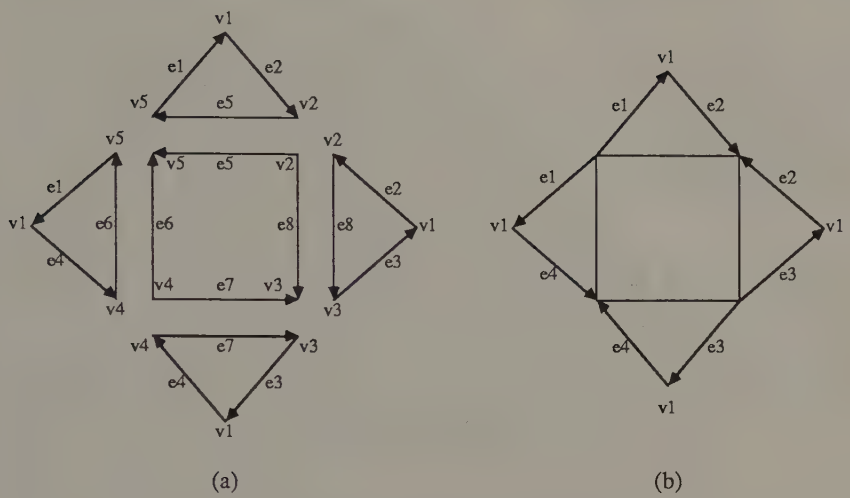



**Figure 3.7** Plane models of a pyramid.

For simplicity, we shall often draw plane models in a condensed form such as in (b), where some identified edges are drawn just once, and their labels are not included. Similarly, we can draw a collection of identified vertices as a single point. Note that Figure 3.7(a) and (b) both represent the same structure $\{N, A, R\}$, and hence the same plane model. We shall also include the infinite "surrounding" polygon in a drawing of a plane model, when convenient (see, for instance, Figure 3.9 on page 44).

## 3.5.5 Realizable Plane Models

We can finally turn our attention to the necessary conditions for the realizability of a plane model.

### Surface Subdivisions

Topological identification as defined above can produce neighborhoods that are not disks and do not, hence, satisfy the definition of a 2-manifold. To exclude these cases, we restrict the topological identification of plane models:

**Definition 3.9** *A plane model is a* (surface) subdivision *if the following conditions are obeyed in the topological identification of its edges and vertices:*

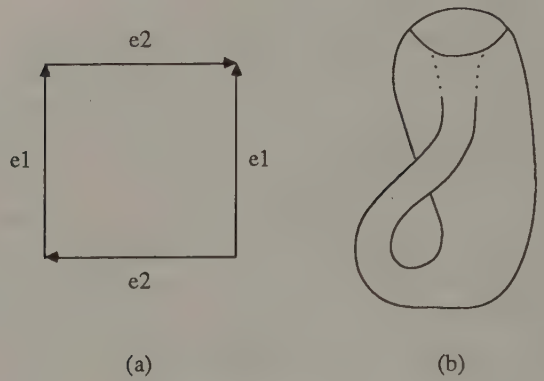



**Figure 3.8** Klein bottle.

1. *Every edge is identified with exactly one other edge.*
2. *For each collection of identified vertices, the polygons identified at that collection can be arranged in a cycle such that each consecutive pair of polygons in the cycle is identified at an edge adjacent to a vertex from the collection.*

Observe that the edge $v2$–$v5$ in Figure 3.7(b) actually represents the pair of edges labeled $e5$ in (a); hence, $v2$–$v5$ satisfies the first condition. The condition disallows objects such as that depicted in Figure 3.4(b); in this case, four edges are identified.

The second condition guarantees that the combined neighborhood of each identified group of vertices is a disk; for instance, the six polygons identified in Figure 3.4(a) cannot be arranged in a single cycle.

### Orientability

There are 2-manifolds that do not have physical counterparts in $E^3$, i.e., that cannot be constructed in three-dimensional space at all, and are hence not the boundary of any $r$-set. For instance, the *Klein bottle* represented by the plane model of Figure 3.8 is such a surface. These nonrealizable 2-manifolds can be distinguished from realizable ones by the concept of *orientability*:

**Definition 3.10** *A plane model is* orientable *if the directions of its polygons can be chosen so that for each pair of identified edges, one edge occurs in its positive orientation in the direction chosen for its polygon, and the other one in its negative orientation.*

This condition is known as the *Möbius' rule.*

In the following, we shall be interested in orientable plane models only as they correspond with surfaces that can be realized in $E^3$, i.e., can form the boundary of an $r$-set. As seen from the definition above, that all polygons are *consistently oriented* (i.e., all polygons are oriented, say, clockwise except the surrounding polygon which is oriented counterclockwise) is sufficient to guarantee the realizability of a plane model.

Intuitively, realizable plane models can be drawn on an $r$-set without crossing edges. A more rigorous way to express this is to note that points and open sets of the plane model can be mapped on points and open sets of the $r$-sets with a continuous transformation. In practice, we prefer mappings that can be represented by assigning geometric information (such as coordinate values, curve equations, and surface equations) to vertices, edges, and polygons of the plane model.

### 3.5.6 The Euler Characteristic

Consider the cube depicted in Figure 3.9(a). The cube has a total of $f = 6$ faces, $e = 12$ edges, and $v = 8$ vertices (where identified edges and vertices count as one). Observe that the plane model includes the surrounding exterior polygon, and that all edges are identified.

Clearly, the same surface could be modeled in terms of the modified model (b), obtained by replacing one square face of (a) by two triangles. The new model has one additional face and edge, or a new total of $f' = 7$ faces and $e' = 13$ edges.

Both plane models represent the same surface—namely, a surface topologically equivalent to the sphere. A property of fundamental importance is that the plane models are directly able to tell us this, as shown by the following theorem:

**Theorem 3.11** *Let the surface $S$ be given as a plane model and let $v$, $e$, and $f$ denote the numbers of vertices, edges, and faces in the plane model. Then the sum $v - e + f$ is a constant independent of the manner in which $S$ is divided up to form the plane model. This constant is called the* Euler characteristic *of the surface and is denoted $\chi(S)$.*

For our purposes, this *invariance theorem* is one of the central results of algebraic topology. Its formal proof would bring us too far from the scope of this book; see e.g. [52]. To get some reliance on it, it is easy to verify for the original cube that

$$\chi = v - e + f = 8 - 12 + 6 = 2.$$

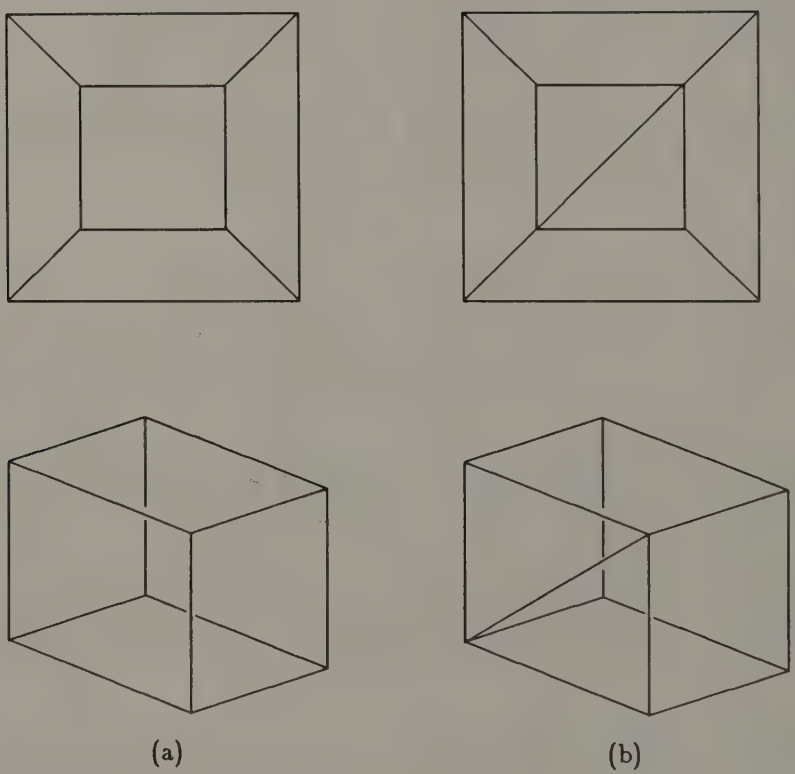



**Figure 3.9** Models of a cube.

After the modification we have

$$\chi' = v - e' + f' = 8 - 13 + 7 = 2,$$

as expected. No matter how we alter the plane model to yield a new plane model, the Euler characteristic remains the same.

### Betti Numbers

The theory of *homology* tells us that the Euler characteristic can be expressed as

$$\chi = h_0 - h_1 + h_2 \tag{3.1}$$

where $h_0$, $h_1$, and $h_2$ are called the *Betti numbers* of the plane model. Formula 3.1 is called the *Euler-Poincaré formula.*

The Betti numbers can be calculated through group-theoretic techniques from the plane model; however, this is beyond the scope of our interest. Instead, let us take a look at the topological significance of the Betti numbers. The second Betti number $h_2$ reveals the orientability of the surface, and equals $h_0$ for the kinds of surfaces we are interested in.

An arbitrary surface is always the union of a number of connected pieces called *components.* The zeroth Betti number equals the number of components. Hence, for the cube $h_0 = 1$, while for a set of six tennis balls $h_0 = 6$.

The first Betti number $h_1$ is called the *connectivity number* of the surface. It tells the largest number of closed curves that can be drawn on the surface without dividing it into two or more separate pieces. For the cube (or any surface topologically equivalent to it) $h_1 = 0$, because every closed curve would cut a part of the surface away. More intuitively, $h_1$ equals twice the number of "holes" in the object. A doughnut has one hole, and its $h_1 = 2$. Hence the Euler characteristic of a doughnut (or a coffee cup) is

$$\chi = 1 - 2 + 1 = 0.$$

The invariance theorem can be generalized for the Betti numbers as well. That is, no matter how a surface is divided up to make a plane model, the Betti numbers (and all topological characteristics conveyed by them) will remain invariant. This fact will be of interest to us in Chapter 9; for reference, let us state it as a theorem:

**Theorem 3.12** *Let the surface $S$ be given as a plane model. Then the Betti numbers $h_0$, $h_1$, and $h_2$ of the plane model are constants independent of the manner in which $S$ is divided up to form the plane model.*

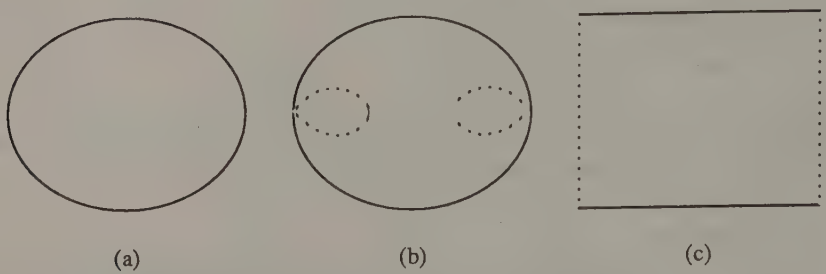



**Figure 3.10** The making of a cylinder.

In the following, we shall use a more mnemonic notation for the Euler-Poincaré formula 3.1. We shall write $s$, the number of connected surfaces or "shells", to denote $h_0$. Similarly, $h$, the *genus* of the surface (or the sum of the genera of all components of a multicomponent model), is used to denote the number of holes and is defined $h_1/2$. With this notation, the Euler-Poincaré formula may be written in more familiar terms as

$$v - e + f = 2(s - h). \tag{3.2}$$

### 3.5.7 Surfaces With Boundary

As seen in previous sections, orientable 2-manifolds coincide with the informal concept of "closed physical surfaces." Sometimes, however, we would like to handle also surfaces that do not necessarily bound a solid.

One such case, the cylinder surface, was already depicted in Figure 3.5. Its plane model was characterized by some edges that were not identified to some other edge. These edges form the boundaries of the surface; hence the term *surfaces with boundary.*

Surfaces with boundary can be thought of as created from their closed counterparts by cutting some polygons away. For instance, a cylinder is created by removing two disks from a sphere, as depicted in Figure 3.10. In the figure, we start from a plane model of a sphere (a). The plane model is modified by removing two polygons (b) which leads to the plane model of the cylinder (c).

Using this mental tool, the theory of closed surfaces can be carried over to surfaces with boundary easily. For instance, to calculate the Euler characteristic of a surface with $b$ boundary components, we proceed by adding $b$ new polygons to its plane model that "mend" it to form the corresponding

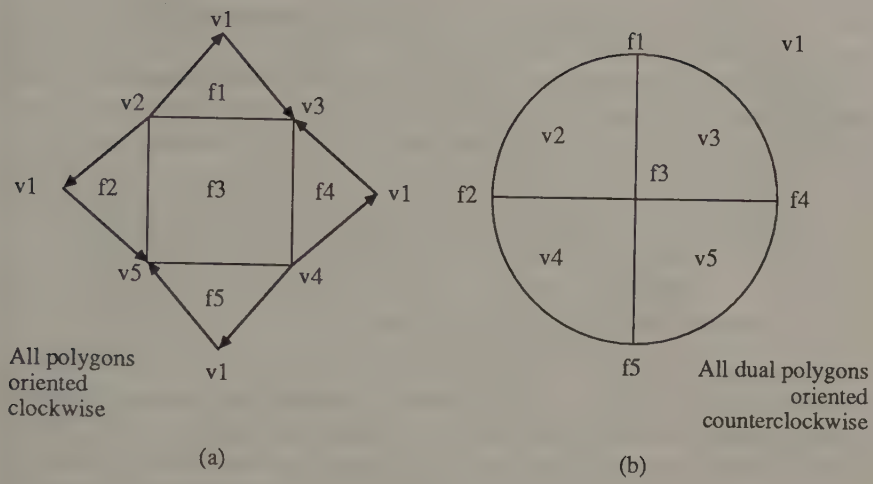



**Figure 3.11** A plane model and its dual.

closed surface. Clearly, the Euler characteristic of the mended surface is

$$\chi = v - e + f + b.$$

Hence for a surface with $b$ boundary components, the Euler-Poincaré formula 3.2 can be written as

$$v - e + f = 2(s - h) - b. \tag{3.3}$$

### 3.5.8 Duality

The *dual* of a planar graph is constructed by assigning a dual vertex to each polygon of the original graph, and joining a pair of dual vertices with a dual edge, if the corresponding original polygons share an edge (possibly through identification). In this fashion, dual polygons correspond with original vertices.

Using this procedure, we can convert a plane model of a surface to its dual plane model. An example of a plane model and its dual is given in Figure 3.11. Observe that the dual model (b) includes the infinite "surrounding" polygon $v_1$.

Note that polygons and edges of the dual plane model are oriented. To explain how the orientation of the dual plane model is chosen, let us

assume that the original model is consistently oriented. First, we use the convention that each dual edge is oriented towards the dual vertex that corresponds with the original polygon in which the edge occurs in its positive orientation. This determines uniquely the orientation of every dual edge. Directions of dual polygons are chosen by the following rule:

> *Let $v$, $e$, $v^*$, $e^*$ denote an original vertex, an original edge adjacent to $v$, the dual polygon corresponding with $v$, and the dual of $e$. If $v$ is the final point of $e$, the orientation of $v^*$ is chosen so that $e^*$ is traversed in its positive orientation, otherwise the opposite orientation is chosen.*

Under this rule, the orientation of dual polygons will be consistent. Furthermore, the dual of a dual plane model will be identical with the original model. We actually define the cyclical ordering of the edges around a vertex to be equal to the ordering of the corresponding dual edges in the dual polygon of the vertex. The dual is a surface subdivision because all its edges are identified with exactly one other edge.

We are interested on duality primarily because of the following:

**Lemma 3.13** *All topological properties of a plane model are preserved in its dual.*

In particular, the Betti numbers and the Euler characteristic $\chi$ of a plane model and its dual are equal.

Duals of plane models will be of use when arguing about the manipulation of plane models in Section 9.1.1.

### 3.5.9 Wrap-Up of Plane Models

We may now conclude the somewhat lengthy development as follows:

> The surface of a solid can be rigorously modeled in terms of a planar graph with a special topology. Based on the graph, we can decude the overall topological properties of the surface, such as its orientability, connectivity, and the number of "holes" through it. Such properties are compactly expressed in the form of the Euler-Poincaré formula.

Of course, for the purposes of solid modeling, we shall be primarily interested in realizable plane models—that is, plane models that can be drawn on the boundary of an $r$-set. As we shall see later, the theory of plane models will allow the derivation of manipulation operations of plane models that always will yield realizable plane models while being general enough to model all plane models of interest.

## 3.6 REPRESENTATION SCHEMES

Having now tackled the proper definition of a mathematical modeling space for solid modeling, we can now consider the problems of assigning computerized representations of the objects of the modeling space. As mentioned in Section 3.2, the three-level view of modeling allows us to precisely define the relationship of objects and their representations.

Let $M$ denote a mathematical modeling space of objects. In this section, let "solid" denote an entity of $M$.

**Definition 3.14** *A* solid representation *is a finite collection of symbols (of a finite alphabet) that designates a solid of* $M$.

**Definition 3.15** *The representation techniques of a given solid modeler define the* representation space $R$ *of the modeler. Those representations that actually can be constructed by the solid modeler according to its syntax rules are termed* admissible. *In this section, we shall call entities of* $R$ *"representations."*

Observe that any solid representation $r \in R$ is meaningful only when related to an entity $m \in M$. Hence the significance of solid representations is expressed by a mapping from $R$ to $M$:

**Definition 3.16** *A* representation scheme *is a relation* $s : M \rightarrow R$. *The domain of* $s$ *(i.e., those* $s \in M$ *that have an image under* $s$*) is denoted by* $D$ *and the image of* $D$ *under* $s$ *by* $V$. *The inverse relation of* $s$ *is denoted by* $s^{-1}$. *That* $r \in R$ *is the image of* $m \in M$ *under* $s$ *will be denoted by* $\{m, r\} \in s$.

The validity of a representation can now be expressed more rigorously:

**Definition 3.17** *Any representation* $r \in V$ *is termed* valid.

Note that each valid representation is hence the image of at least one entity of $M$. However, we neither assume that all solids of $M$ are representable (i.e., $D$ need not equal $M$) nor that all representations of $R$ are valid (i.e., $V$ need not equal $R$). See Figure 3.12 for an illustration of this.

With the concepts developed so far, we can characterize ambiguity as a property of a representation scheme:

**Definition 3.18** *If any valid representation models exactly one solid under* $s$ *(i.e., it is the image of exactly one solid of* $M$*),* $s$ *is called* unambiguous *or* informationally complete. *In other words, this property requires that for each valid representation* $r \in V$, *it holds that*

$$\{m, r\} \in s \wedge \{m', r\} \in s \Rightarrow m = m'.$$

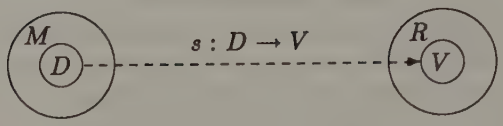



**Figure 3.12** Definition of a representation scheme.

Unambiguity assures that a representation always designates a unique solid. Hence unambiguous representations in principle contain complete information of the solid. Note that a solid may still have many representations under an unambiguous representation scheme. A symmetric variation of unambiguity requires that this may not happen:

**Definition 3.19** *A representation scheme $s$ is termed* unique *if all solids $m \in D$ have exactly one representation, i.e., if*

$$\{m, r\} \in s \wedge \{m, r'\} \in s \Rightarrow r = r'.$$

If interpreted literally, uniqueness requires that a solid has just a single representation independent of its orientation. As this is unfortunately rarely the case, we shall also use a weaker form of uniqueness as defined below:

**Definition 3.20** *A representation scheme $s$ is* weakly unique *if every oriented instance of every solid $m \in D$ has only one representation.*

In the following, the term "uniqueness" will refer to weak uniqueness, unless explicitly mentioned otherwise.

### 3.6.1 Properties of Representation Schemes

Alternative major approaches to solid modeling differ in the modeling spaces, representation spaces, and representation schemes they can support. In addition, the definition of representation schemes allows us to

discuss their desired properties on a fairly general level. Some properties were already described in the preceding section. For clarity, the following list rephrases those properties as well.

1. *Expressive power:* What objects are included in the domain $D$ covered by the representation scheme? Is it possible to extend the domain? One aspect of the expressive power is the *precision* of the representation scheme: how accurately can complicated objects be modeled?

2. *Validity:* Are all admissible representations valid, i.e., do they designate some solids of $M$? A scheme with this desirable property is termed *syntactically valid* as it enforces validity of all solid descriptions that obey its syntax rules.

3. *Unambiguity and uniqueness:* Do all valid representations model exactly one solid? Do some solids have more than one valid representation?

4. *Description languages:* What kinds of solid description languages can be supported in a modeling system based on the representation scheme? Are they *self-contained*, i.e., directly based on the representations of the scheme, or are they based on a *conversion* from other representations?

5. *Conciseness:* How large (in terms of computer storage) do representations of practically interesting solids become? (This property is often in contradiction with the precision of the representation.)

6. *Closure of operations:* Do solid description and manipulation operations preserve the validity of solid representations? How general are the manipulations that can be supported?

7. *Computational ease and applicability:* What kinds of algorithms can be written for the representations of the scheme for, say, computations of Table 1.1? What kinds of computational complexities are involved? What kinds of applications are the representations of the scheme suited for? Of course, we are particularly interested on "self-contained" solutions that do not involve a conversion into some entirely different representation.

Some of these properties have a formal meaning in terms of the theory of representation schemes, while others have a practical flavor which is important in implementing a particular modeling system.

## 3.7 PRIMITIVE INSTANCING

Let us now clarify the theory of the preceding section by looking at a sample scheme for representing solid objects. This scheme will be of interest also in later chapters.

### 3.7.1 Concepts of Primitive Instancing

The simplest way of describing geometric objects is the *Primitive Instancing* scheme. In this approach, we restrict ourselves to modeling objects belonging to one of predefined primitive object types. For instance, we might model mechanical components by generating a sufficiently large collection of "library" parts. Models are simply instances of them, created with an instantiation operation. The instantiation requires a few descriptive parameters (such as, say, component type and size) that together form the model. Algorithms for this scheme operate by examining a tuple of descriptive parameters, and performing operations tailored for each component type.

Consider, for instance, an instancing scheme consisting of the single primitive type "t-brick." Its descriptive parameters would consist of an object type code, and five numerical values $l$, $h_1$, $h_2$, $w_1$, and $w_2$ indicating the dimensions of the object as depicted in Figure 3.13. Additionally, the orientation of the brick would be specified, say, in terms of a rigid transformation.

The domain $D$ of this instancing scheme is the set of all bricks, and the representation space $R$ consists of all 6-tuples (records) of the form

$$(tbrick, l, h_1, h_2, w_1, w_2).$$

To be a valid representation, i.e., to designate a physically meaningful object, the parameters should satisfy the following inequalities:

$$\begin{aligned} 0 &< l \\ 0 &< h_1 \\ 0 &< h_2 \leq h_1 \\ 0 &< w_1 \\ 0 &< w_2 \leq w_1. \end{aligned} \tag{3.4}$$

Obviously, primitive instancing schemes are unambiguous: a valid tuple fully determines a brick. Their uniqueness is somewhat more problematic, however. Too see why, let us introduce another primitive into our sample instancing scheme, the primitive type "block," described with a type code and numerical values $l$, $h$, and $w$ (Figure 3.14). Observe that two different

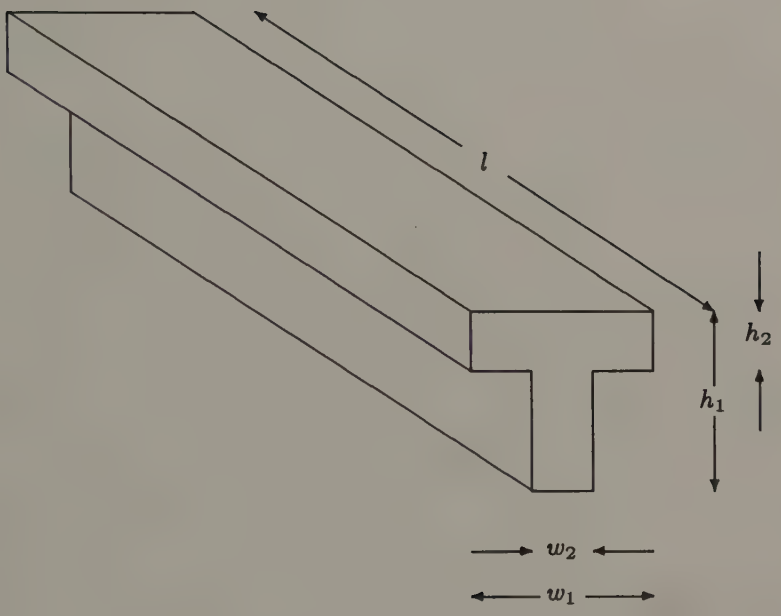



**Figure 3.13** T-brick.

ENGINEERING AND SCIENCE LIBRARY,
P.C.L., 116 NEW GATE STREET, LONDON, WC1A 3AB

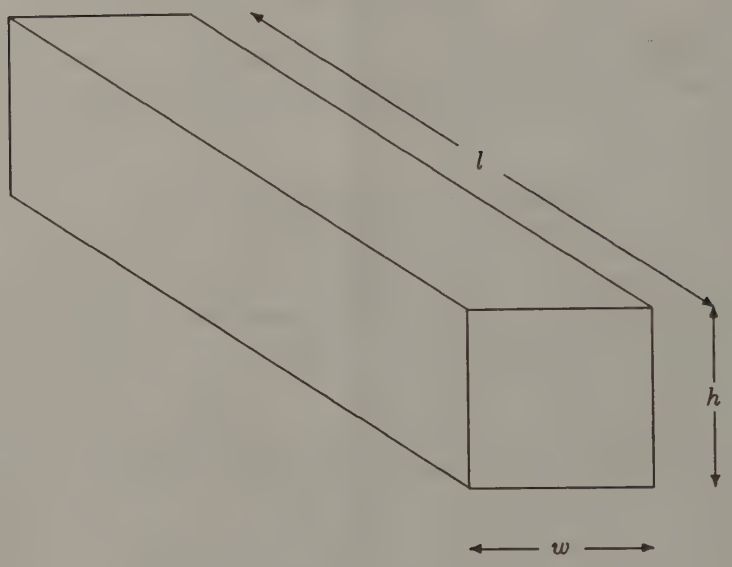



**Figure 3.14** Block.

tuples

$$(tbrick, l, h, h, w, w)$$

and

$$(block, l, h, w)$$

describe the same object (Equations 3.4 allow $w_1 = w_2$). We may conclude that primitive instancing schemes are not necessarily unique.

### 3.7.2 Properties of Primitive Instancing

Let us discuss the properties of the primitive instancing scheme according to the framework given in Section 3.6.1.

*Expressive power:* Clearly, the modeling space of a primitive instancing scheme is determined completely by (1) the collection of primitive object types and (2) the nature of the instantiation operation. Obviously, it is painful to incorporate a large modeling space in this approach. A more involved problem is that it is not easy to find natural parameterizations for certain parts.

*Validity:* It is easy to build validity enforcement checks into the instancing operation. For instance, in the example above this amounts to checking whether the Equations 3.4 are satisfied.

*Unambiguity and uniqueness:* Instances always determine uniquely an object; hence they are unambiguous. As the example shows, they are not unique.

*Description languages:* The essential task describing an object in the primitive instancing scheme is the *selection* of the proper solid type from a fixed collection. The selection can be performed in many ways, including a menu driven, graphical operation. These procedures are obviously self-contained.

*Conciseness:* Properly structured primitive instancing schemes are concise.

*Closure of operations:* Primitive instancing does not support object modification operations in the general sense: the only possibility is the modification of instantiation parameters. As this operation can be subjected to similar checks as the original instantiation, the operations of the scheme are closed, i.e., yield valid models.

*Computational ease and applicability:* Algorithms for the primitive instancing scheme take the form of a (huge) case analysis, that finally breaks into tailored code for each object type. In the example, the volume of an object would be calculated by examining the type code and by performing a different routine for each object type. The resulting computational power can be good, because each case can be separately optimized.

As we shall see in the following chapters, primitive instancing is often used as an *auxiliary* scheme in modelers based on other representations in order to ease the description of often-needed parts. In this role it is indispensable, for instance, in problem areas such as design of electrical instrumentation or piping.

## 3.8 A TAXONOMY OF SOLID MODELS

As set forth in the preceeding sections, we may view solid objects as point sets of the Euclidean three-dimensional space satisfying restrictions that catch our idea of "solidity." To such sets, solid modeling aims to assign finite representations suitable for generating data for algorithms.

Primitive instancing can be viewed as a method of describing certain point sets indirectly through a tuple of their attributes. Unfortunately, as we saw in the previous section, either the modeling space of the modeler must be severely limited or a excessively large collection of primitives must be supported to make this approach generally applicable.

Therefore, we need a representation that encodes the infinite point set in a finite amount of computer storage in a more generic fashion. Representations achieving this may be divided in three large classes as follows:

1. *Decomposition models*, that represent a point set as a collection of simple objects from a fixed collection of primitive object types, combined with a single "gluing" operation. (Primitive instancing can be considered a special case of this approach.)

2. *Constructive models*, that represent a point set as a combination of primitive point sets. Each of the primitives is represented as an instance of a *primitive solid type*. Constructive models include more general construction operations than mere gluing. As we shall see, their theory is based on the point-set formulation of solidity presented in Section 3.4.

3. *Boundary models*, that represent a point set in terms of its boundary. The boundary of a three-dimensional "solid" point set is a two-dimensional surface that is usually represented as the collection of *faces*. Faces, again, are often represented in terms of their boundary being a one-dimensional curve. Hence boundary models may be viewed as a hierarchy of models. Boundary models are based on the surface-oriented formulation of solid objects presented in Section 3.5.

These major approaches to solid modeling will be described in the following three chapters. We shall also be interested on *multirepresentational* and *hybrid* modelers that employ several representations simultaneously. These models shall be dealt with in Chapter 7.

## PROBLEMS

3.1. The footnote on page 32 points out that sometimes we are not modeling physical reality, but an idealized world. Can you think of other examples of this besides engineering design?

3.2. Discuss the physical meaningfulness of nonmanifold objects such as those of Figure 3.4 on page 37. Are some more "meaningful" than others? Can you think of criteria that would allow us to make a distinction between "meaningful" and "nonmeaningful" nonmanifold objects?

3.3. The nonuniqueness of primitive instancing schemes can be demonstrated without introducing the "block" primitive in addition to the "t-brick." Give two different "t-brick" 6-tuples that correspond with the same object.

## BIBLIOGRAPHIC NOTES

Much of the material of this chapter is based on the works of Aristides A.G. Requicha and his colleagues with the PADL-project at the University of Rochester. The technical report series of the PADL-project is strongly suggested reading to all persons interested on the fundamentals of solid modeling.

The three-level modeling approach is originally presented in report [94] that also discusses in detail point-set and manifold models. The report goes into much more detail as for the group-theoretic techniques for dealing with plane models, although Requicha uses a different notation.

Report [97] can be considered a basic reference to point-set models and constructive models which are the subject matter of Chapter 5.

The article [95] draws some of the material of the technical reports cited above in a more streamlined form, and presents the theory of representation schemes. The list of desired properties of solid presentations of Section 3.6.1 is based on the list presented in this article.

Plane models for rigorous discussion on solid models were introduced in articles by Hanrahan [50] (briefly) and this author [75] (extensively). In their article [49], Guibas and Stolfi present another variation of plane models, and modeling primitives that can handle even nonorientable objects such as the Klein bottle.

Texts on topology and particularly in algebraic topology are usually rather heavy for a non-mathematician reader. The (in this chapter often-cited) book of Henle [52] is an exception of this rule. Other standard texts include [3,1,30]. A standard reference in graph theory is [51].

# Chapter 4

# DECOMPOSITION MODELS

Decomposition models describe solids through a combination of some basic building blocks glued together. The kinds on basic objects used and the way the combination of basic objects is recorded leads to variations on this basic theme.

## 4.1 EXHAUSTIVE ENUMERATION

Earlier, we viewed solids as continuous point sets. While we cannot list all points belonging to a solid, we can easily list all, say, tiny cubes that are contained (completely or partially) in the solid. The cubes are assumed to be nonoverlapping and of uniform size and orientation, i.e., they form a *regular subdivision* of the space.

The resulting solid representation is called the *exhaustive enumeration*; see Figure 4.1 [28]. In the collection of cubes forming the exhaustive enumeration, each small cube can be completely described in terms of its corners. By regularity, it is sufficient to store the coordinates of just one corner for each cube of the collection. In the case that we can *a priori* limit our interest to some *space of interest* being a subset of $E^3$, the collection can be efficiently encoded as a three-dimensional array $c_{ijk}$ of binary data. The array represents the "coloring" of each cube: if $c_{ijk} = 1$ ("*black*"), the cube $ijk$ represents a solid region of the space, and otherwise it is empty ("*white*").

The binary array representation is, of course, the obvious way to represent a picture in *digital image processing*. Reference [112] surveys object

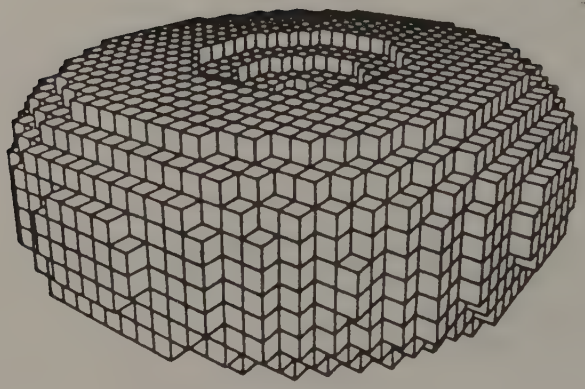


**Figure 4.1** Exhaustive enumeration.

representations of this field; see also [137]. While image processing as such is beyond the scope of this book, the areas of interest of image processing and solid modeling do intersect, and much of what we shall say on the processing of exhaustive enumerations (or decomposition models in general) has its roots in image processing techniques.

### 4.1.1 Construction of Exhaustive Enumerations

An independent solid description mechanism for exhaustive enumerations would consist of a list of all cubes that are considered to form parts of the solid. In image processing applications this is indeed possible with image scanning devices. Digital tomography [123] can also supply three-dimensional digital information.

Unfortunately, in solid modeling we rarely have an image or an object to start with, and these direct solid description techniques cannot be used. It would also be very hard to describe realistic objects as directly enumerating its cubes. A more practical approach is to create exhaustive enumerations by a conversion from some other representation such as the constructive or boundary models to be discussed in the subsequent chapters. Other objects can then be created by Boolean set operations or other suitable algorithms for exhaustive enumerations.

The interesting point of exhaustive enumeration is that it is straightforward to produce algorithms that create new representations from existing representations. A prime example is the calculation of Boolean set oper-

ations on exhaustive enumerations. In the binary matrix representation, the algorithm becomes just the corresponding bitwise operation for all corresponding elements.

Simplistically, a binary matrix is always a valid solid representation; hence, all algorithms that produce them can be considered closed. However, often disconnected cells that do not have any neighbor cells are not desired, and algorithms such as the Boolean set operations must pay attention to disallowing them. Quite similarly, a single white cell within a block of black cells might not be considered correct. For various situations, various degrees of connectivity may be desired. Each 3-dimensional cell has six face neigbors, 12 edge neighbors, and eight vertex neighbors. The strictest connectivity criterion would require that all black cells have at least one black face neighbor.

Finding and filling connected components (with various degrees of connectivity) are standard image processing algorithms [92]. It might be possible to apply image processing techniques for solid modeling purposes also more generally. For instance, a *growing algorithm* (*dilatation*) could be used to calculate the "offset solid" of a given solid, i.e., the solid that contains all points at distance $d < r$ from any point of the original solid. This operation would be useful for the generation of data for numerically controlled (NC) machine tools and robotics.

### 4.1.2 Uses of Exhaustive Enumeration

Two- and three-dimensional variants of the exhaustive enumeration are the obvious representations for digital images, and are widely used in digital image processing. Through this connection, exhaustive enumeration has found its solid modeling uses in special applications, such as modeling of cell particles. In this area, a cube is allowed to have other "colors" beside mere "white" or "black," again raising the memory consumption.

Exhaustive enumeration is as much a representation of empty spaces as of spaces occupied by material. A coloring scheme can be used to distinguish various kinds of empty spaces; this is useful, for instance, for modeling buildings in heat transfer analysis. In this context, empty spaces in the interior and in the exterior of the building must be modeled differently.

Exhaustive enumeration is also used as an auxiliary scheme for speeding up operations on other representations. For instance, while being based on the half-space approach to be described in the next chapter, the solid modeler TIPS [91] uses a regular three-dimensional grid as a geometric directory to the elements of the half-space model in order to speed up various geometric algorithms. We shall return to this in Chapter 18.

### 4.1.3 Properties of Exhaustive Enumeration

In summary, let us discuss the properties of the exhaustive enumerations according to the framework of Section 3.6.1.

*Expressive power*: Exhaustive enumeration is obviously *approximative*: surfaces that are not coplanar with any of the coordinate planes of the subdivision will be only approximately represented. With this restriction, however, the scheme is general: all kinds of objects can be represented under it.

*Validity*: The validity of exhaustive enumerations depends on the geometric interaction of individual cubes. Specifically, each pair of blocks may intersect only at a common vertex, edge, or face. If the binary matrix representation can be used, all exhaustive enumerations are valid if connectivity is not required.

*Unambiguity and uniqueness*: All valid exhaustive enumerations are unambiguous. They are also *unique*: in a fixed space of interest and resolution, each solid has just one representation.

*Description languages*: In image processing applications, exhaustive enumerations are generally created through scanning of an image. In the context of solid modeling proper, their generation is ordinarily based on conversion from other models.

*Conciseness*: Exhaustive enumerations tend to be fairly large: a resolution of $256^3$ (which is only barely adequate) takes 16 million bits of storage. Storage requirements rise sharply with increased resolution.

*Closure of operations*: Exhaustive enumerations support many closed algorithms. Boolean set operations are a prime example.

*Computational ease and applicability*: Algorithms for this scheme tend to be extremely simple; however, the mere size of the object representations means that they are slow in an ordinary computer. Note, however, that the computation required is typically extremely *regular*: the calculation needs to consider just one cube or at most the cube and a few neighbor cubes. This regularity means that these representations are prime candidates for VLSI implementation. This can radically affect the applicability of exhaustive enumeration in the future.

## 4.2 SPACE SUBDIVISION SCHEMES

Exhaustive enumerations have many virtues: they are simple, general, and allow the use of a wide variety of algorithms. These good points are, however, offset by the huge memory consumption and the mediocre accuracy possible. To overcome this, many representations replace the underlying

regular space subdivision of the pure enumeration by a more efficient, adaptive subdivision.

These *adaptive subdivision schemes* are based on the simple observation that in the three-dimensional grid of an exhaustive enumeration, the neighbor cubes of a white cube are very likely to be white as well. By encoding this information into one combined node of the data structure considerable savings over the complete grid are possible.

Adaptive subdivisions use the fundamental property that the number of nodes needed for the representation of a solid is proportional to its surface area [82]. Hence the number of elements is proportional to the square $r^2$ of the resolution $r$ used, whereas the size of exhaustive schemes is proportional to $r^3$.

### 4.2.1 The Octree Representation

Prime examples of adaptive space subdivision schemes are the *Octree representation* for solid objects [60,82] and the analogous *Quadtree representation* [106] for two-dimensional objects.

The octree representation uses a recursive subdivision of the space of interest into eight *octants* that are arranged into an 8-ary tree (hence the name). Figure 4.2(a) depicts the octant subdivision.

Usually the octree is thought of as being located around the origin of its local $xyz$-coordinate system, with its first-level octants corresponding to the octants of that space, and in particular octant 3 being the positive octant $x, y, z > 0$ of the space. Hence it is necessary to represent the actual space of interest separately in terms of an appropriate transformation.

Each node of an octree consists of a *code* and eight pointers towards eight sons, numbered 0 through 7. If *code* = *black*, the part of the space represented by the node is all material and the pointers 0 through 7 are nil, i.e., the node is a leaf. If *code* = *white*, the part of the space is empty and the node is again a leaf. The third possibility *code* = *grey* corresponds to the case where the part of the space is partly material and partly empty. In this case, the eight pointers point to eight children that correspond to a regular subdivision of their parent node. For instance, the object of Figure 4.2(b) would be represented by the 2-level octree shown in (c).

Program 4.1 outlines a data structure for representing octrees. Here the space of interest is an orthogonal box of the $xyz$-space, represented by a special "root" node.

It is easy to show that 7/8 of the nodes of an octree are leaves. Because of this, also leaves are usually represented with a special node to avoid allocating storage for the eight nil pointers.

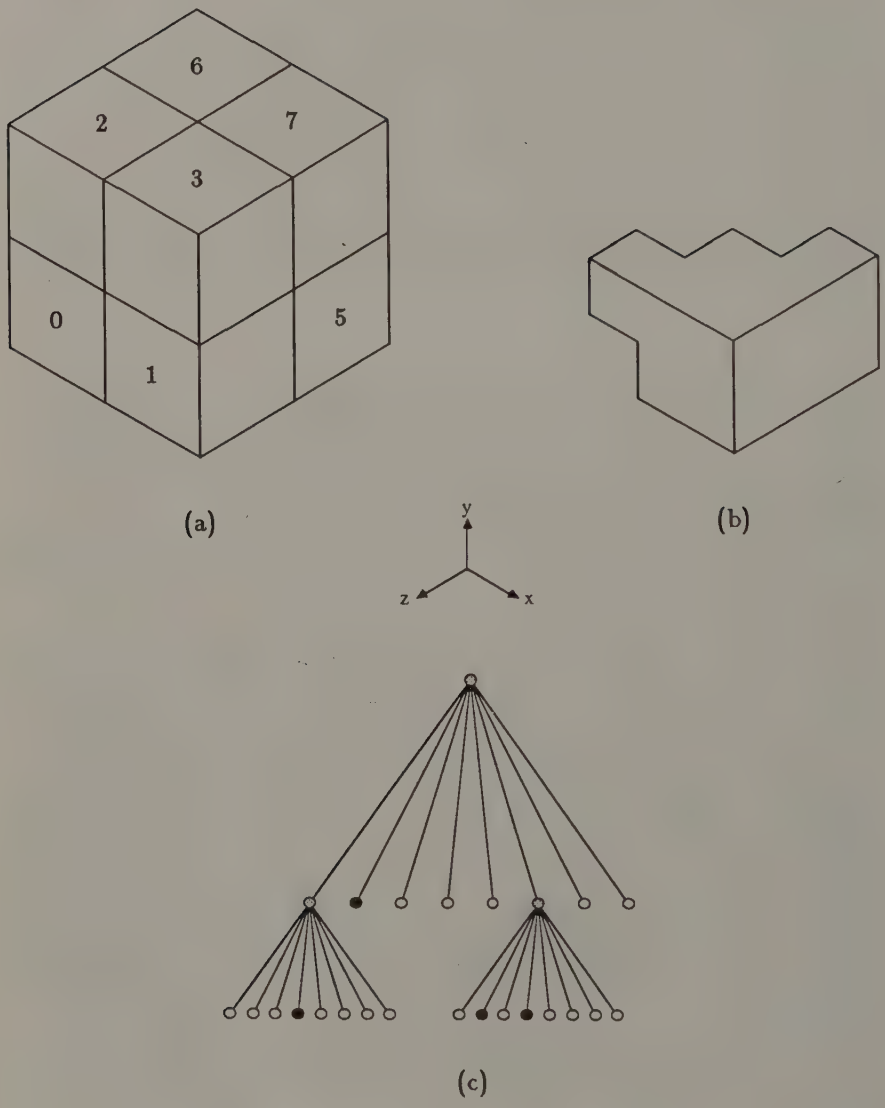



**Figure 4.2** An octree model.

```
struct octreeroot
{
        float           xmin, ymin, zmin;  /* space of interest */
        float           xmax, ymax. zmax;
        struct octree   *root;             /* root of the tree */
};

struct octree
{
        char            code;              /* BLACK, WHITE, GREY */
        struct octree   *oct[8];           /* pointers to octants,
                                              present if GREY */
};
```

**Program 4.1** Octree data structures.

### Construction of Octree Representations

In the solid modeling context, octrees are ordinarily constructed from solid primitives. For each primitive type, a *classification procedure* between an instance of the primitive and an arbitrary node of the octree is needed. The classification must be capable of distinguishing between the following three cases:

1. The node considered is completely in the exterior of the primitive.
2. The node considered is completely in the interior of the primitive.
3. The node is partially in the interior of the primitive.

The classification procedure is used in a recursive fashion. Initially, the whole space of interest is represented by one node with *code* = *white*, and the algorithm is started at the single white node. Each node is classified against the primitive with the classification procedure. If the first or the second case above applies, the node is marked *white* or *black*, and the recursion terminates. Otherwise, the algorithm proceeds by marking the node *grey*, subdividing it into eight octants, and calling itself recursively for each octant. The subdivision is continued until a desired resolution has been reached, usually up to 6–12 levels.

The construction algorithm is outlined in Program 4.2. The actual solid primitives could be represented according to the primitive instancing scheme, i.e., in terms of a primitive type code and dimension and orientation parameters.

```
make_tree(p, t, depth)
primitive *p; /* p = the primitive to be modeled */
octree *t;    /* t = node of the octree, initially
                 the initial tree with one white node */
int depth;    /* initially max. depth of the recursion */
{
       int    i;
       switch(classify(p, t))
       {
           case WHITE:
               t->code = WHITE;
               break;
           case BLACK:
               t->code = BLACK;
               break;
           case GREY:
               if(depth == 0)
               {
                   t->code = BLACK;
               }
               else
               {
                   subdivide(t);
                   for(i=0; i<8; i++)
                        make_tree(p, t->oct[i], depth-1);
               }
               break;
       }
}

/* classify octree node against primitive */
classify(...)

/* divide octree node into eight octants */
subdivide(...)
```

**Program 4.2** Construction of an octree.

The collection of possible primitives depends on whether a classification algorithm for the primitive can easily be implemented. Typical "easy" primitives include rectilinear blocks, half-spaces such as sphere, cylinder, and cone, and certain higher-order objects such as tori and objects bounded by so-called superquadrics [10].

Octrees and quadtrees can also be constructed from digital image information if such is available.

### Algorithms for Octrees

A functionally complete octree modeler should include algorithms for the following categories of tasks:

1. *Tree generators* that create octrees from parameterized primitives or other types of geometric models.

2. *Set operations* that take two octrees (with identical spaces of interest) and calculate a new octree that gives the Boolean union, intersection, or set difference of the two arguments.

3. *Geometric operations* that take an octree and calculate a new octree that models the result of translating, rotating, or scaling the object modeled. Another type of geometric operation calculates a new octree that has been altered according to the *perspective transformation*. Some of these problems lead to unintuitive and complicated algorithms. For instance, the translation of an octree is a relatively hard operation.

4. *Analysis procedures* that calculate properties such as the volume or the surface area of an octree. A *connected components* procedure that labels each octree node with the identifier of the connected object it belongs to gives an example of a more involved analysis operation.

5. *Display generators* that create a graphical image of the object modeled by the octree.

The generation of octrees was already outlined in the above. In the following, we shall briefly discuss some of the other areas.

**Graphical Output** The essential property of octrees is that they record the shape information of an object in a spatially ordered manner. If this can be exploited in algorithm design, it is possible to create very simple algorithms that just scan their argument trees and perform relatively simple operations at each node.

Generation of graphical output for a frame buffer raster display is a particularly good example of this. Observe that by traversing the nodes of the octree in a suitable order, the parts of the image generated from far nodes can be painted before those of the near ones. Hence the close parts will overlay the far away parts in the frame buffer, and no sorting is required for the generation of an image with hidden surfaces removed.

The proper ordering depends on the location of the viewer with respect to the tree. For instance, if the viewer is located in the octant $x, y, z > 0$ of the space (such as in the view of Figure 4.2(a)), the ordering 4, 0, 5, 1, 6, 2, 7, 3 of the subtrees will produce the correct result. Reference [31] describes techniques for the generation of more advanced images based on this approach.

**Analysis** Many analysis operations can also be performed according to a similar node-by-node paradigm. For instance, integral properties such as volume, center of mass, and moments of inertia can all be calculated by simple tree traversal algorithms.

**Set Operations for Octrees** Also set operations for octrees lead to a tree traversal algorithm. The set operations algorithm takes two argument octrees, and produces a third octree that represents the desired Boolean set operation (union, intersection, set difference) of the arguments.

The argument trees are traversed in a synchronous fashion, and a case analysis as for the operation and the types of the corresponding nodes is performed. For instance, when calculating the set intersection, the following cases would occur in the processing of two corresponding nodes $n_1$ and $n_2$:

1. Nodes $n_1$ and $n_2$ are both leaves. In this case, the corresponding node of the result octree is black if both $n_1$ and $n_2$ are black; otherwise, it is white.

2. Either $n_1$ or $n_2$ is a leaf. In this case, if the leaf node is black, the subtree of the nonleaf is copied to the result octree. Otherwise, the node of the result tree is white.

3. Nodes $n_1$ and $n_2$ are both nonleaves. In this case, the algorithm considers recursively their children as above.

Observe that if the both octrees are already present (i.e., the time needed for their construction can be ignored), the complexity of this algorithm is at most proportional to the size of the smaller tree.

**Limitations of tree traversal algorithms** Unfortunately, not all algorithms for octrees can be brought in the form of a simple tree traversal. In graphics, visual effects such as smoothly shaded surfaces and transparency require the capability of accessing the neighbors of a node. Unfortunately, accessing a neighbor node can in the worst case require a traversal up to the root of the tree, and down to the neighbor. Similar lines apply also to various image processing algorithms such as "growing" and connected component labeling.

The complexity of accessing neighbor nodes can be somewhat cured by introducing further pointers in the octree data structure. As this raises the cost of the generation of octrees, careful analysis of the frequency of the various operations is needed for maintaining the proper balance.

### Properties of Octrees

The properties of octree representations are similar to those of the exhaustive enumeration, with some noticeable exceptions.

*Expressive power*: As were exhaustive enumerations, octrees are approximative representations, and model exactly only rather peculiar objects. However, if a classification routine for a primitive can be written, arbitrarily accurate octrees for it can be generated (at the cost of high storage use).

*Validity*: If no special conenctivity requirement are posed, all octrees are valid representations of some solid.

*Unambiguity and uniqueness*: Up to the limits of resolution, all octrees unambiguously define a solid. On a fixed resolution, the representation is also unique: an object has just one compacted[1] octree representation with at most $n$ levels.

*Description languages*: The same lines apply as for exhaustive enumerations. Octrees are usually formed by conversion from other representations, of which constructive representations have an especially important role. In image processing applications, quad- and octrees are also formed directly from rasterized image data.

*Conciseness*: In general, the number of nodes in the octree representation of a solid object is proportional to the surface area of the object [82]. Hence octree models are not quite as large as exhaustive representations but still take a fair amount of storage. An octree representation of an "average" engineering object easily takes more than 1 million bytes of memory.

[1] Algorithms such as set operations can create octrees with unnecessary nodes (e.g., an internal node whose children are all black). Such nodes can be removed with a relatively simple tree traversal algorithm.

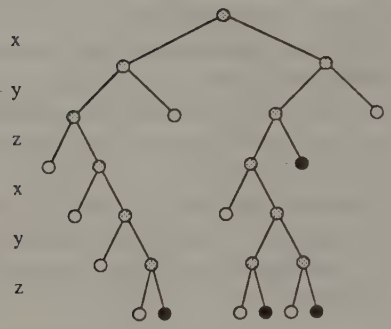



**Figure 4.3** Binary subdivision tree.

*Closure of Operations*: Octree models support closed algorithms for problems such as translation, rotation, and Boolean set operations.

*Computational ease and applicability*: Many algorithms for octrees take the form of a tree traversal where a relatively simple operation is performed at each node of the tree. To these algorithms, the same notes apply as to exhaustive enumeration, including the potential role of VLSI.

Octree and quadtree representations have won considerable interest, and their literature is large and grows rapidly. For a comprehensive survey, the reader is referred to [106].

### 4.2.2 Binary Space Subdivision

As an alternative to the octree formulation, a *binary space subdivision* is also possible. As the name implies, this approach divides a grey node in two halves, instead of eight octants. The subdivision is performed successively in the direction of $x$, $y$, and $z$ (see Figure 4.3).

In comparison with octrees, explicit binary subdivision trees are slightly smaller. As the virtues of the binary subdivision approach are most apparent in linearized versions to be described below, we shall not discuss them further here.

### 4.2.3 Linearized Space Subdivision

While offering significant storage savings over exhaustive enumeration, octrees (and binary subdivision trees) are still quite large. This has led

researchers to investigate possibilities for compressing the octree representation further, and many alternative representations that replace the explicit tree structure with a pointer-free, linear data structure have been proposed. Obviously, such representations are also convenient for storing octrees into external storage.

### Linear Octrees

Recall the numbering 0 ... 7 of octants of the octree. The octant numbers can be used to construct a *path address* for each node of the octree except the root. Clearly, the path address of a octree node of level $i$ becomes a sequence of $i$ digits $0, ..., 7$. A special "end-of-number" digit can be included to mark the tail of a number that has less digits than the maximal resolution.

The *linear octree* of Gargantini [42] is based on these observations. In her approach, the linear octree is simply the sorted list of path addresses to all black nodes. Hence, the linear octree that corresponds to the octree of Figure 4.2(c) is simply the list

$$\{03, 1X, 51, 53\}$$

where $X$ denotes the "end-of-number" digit.

Another compact linear encoding of an octree is the so-called *DF-representation* [63]. It is formed by traversing the octree in preorder, and storing certain information of the nodes encountered. The encoding uses the alphabet "B", "W", and "(" denoting a black leaf, a white leaf, and an internal (nonleaf) node. Hence, the DF-representation of the octree of Figure 4.2(b,c) is the following string:

```
((WWWBWWWWBWWW(WBWBWWWWWW
```

As the alphabet has only three characters, two bits per each node are sufficient for the encoding.

Remarkably, mere linear representations are sufficient for many important algorithms. Boolean set operations on linear octrees end up in merging two linear strings of characters, a relatively straightforward operation. Algorithms for finding the nearest neighbor of a given node, for performing certain geometric operations, and for calculating connected components of objects also have been reported. For instance, reference [108] describes several algorithms for the two-dimensional case.

### Bintree: Linear Binary Subdivision

Binary subdivision schemes lend themselves to very similar compact representations. In comparison with linear octrees, they are slightly smaller

because the number of leaves is smaller (compare Figures 4.2 and 4.3). Also, two leaves at the lowermost level of the tree under same parent can be encoded with one bit only because there are only two possibilities (black-white, white-black). These and other enhancements lead to linear representations having approximately one bit per tree node [115].

In three dimensions, Samet and Tamminen call the resulting solid representation a *bintree* [107,65]. Again, many algorithms can work without ever constructing an explicit tree. For instance, the paper [107] describes an algorithm for the evaluation of Boolean expressions of solid primitives. Display algorithms for bintrees are described by Koistinen *et al.* [65].

### 4.2.4 Geometric Search by Subdivision

Having an explicit octree available, it is very easy to check whether some particular location in the space of interest contains material or not. This point of view leads us to consider octrees and other space subdivision schemes as an efficient access method to geometric information.

Ordinarily, octree nodes only store a few bits of information encoding the material class of the cell. In the case of a geometric index, nodes may store (references to) other geometric entities, such as points, lines, or surfaces.

For the purposes of solid modeling, the prime example of this approach is the *extended octree* [7,22,86] that stores geometric elements of a boundary representation into an octree. As we interpret the extended octree as a hybrid representation scheme, we shall discuss its details in Chapter 7.

## 4.3 CELL DECOMPOSITIONS

Another approach to resolve the problems of exhaustive enumeration while preserving its nice properties is to use also other kinds of basic elements than just cubes. These schemes are called *Cell Decompositions.*

### 4.3.1 Representation of Cell Decompositions

A cell decomposition has a certain variety of basic *cell types* and a single combination operator *glue.* Individual cells are usually created as parameterized instances of cell types. Cells may be any objects that are topologically equivalent to a sphere (i.e., do not contain holes); in particular, they may include curved surfaces. A solid is modeled by means of a collection of semidisjoint cells, i.e., cells may "touch" each other along their bounding surfaces, but not have common interior points; see Figure 4.4 [35].

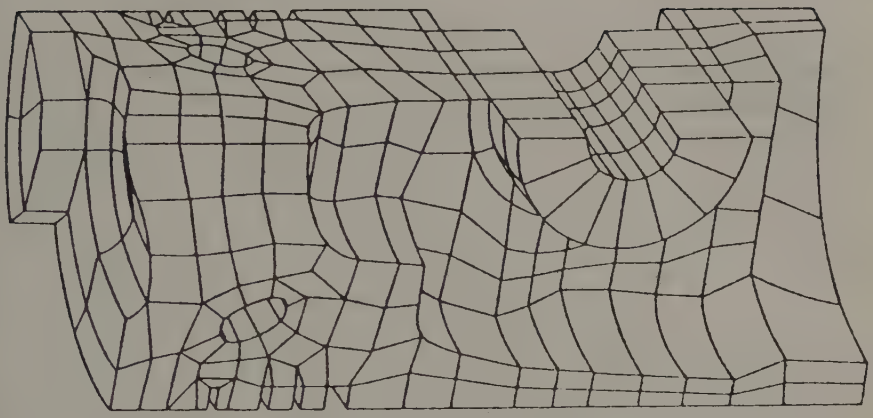


**Figure 4.4** A cell decomposition.

A typical cell might be a curved polyhedron determined by 20 points. Eight points reside at its corners, and 12 points on the lines between the corners. Hence, each line has three points which define a quadratic curve, and each surface becomes a biquadratic surface patch determined by eight points (see Figure 4.5).

Typically, the collection of cells must satisfy also certain topological criteria in addition to being pairwise semidisjoint. Usually, any two cells are required either to be completely disjoint or meet in exactly one corner, along one line, or along one face. The case that two cells intersect otherwise is considered an incorrect cell decomposition. Such requirements make it quite difficult for a human being to construct valid cell decompositions directly.

In the important case of finite element models (FEM), the combination operation *glue* is usually represented implicitly through an encoding of "nodes" in the cell decomposition data structure. This is important for arranging the communication between a cell and its neighbors.

### 4.3.2 Properties of Cell Decompositions

Again, let us discuss the properties of cell decompositions according to the general scheme.

*Expressive power*: The modeling space is general and in the presence of cells with curved surfaces exact up to the degree of the cells, typically quadratic.

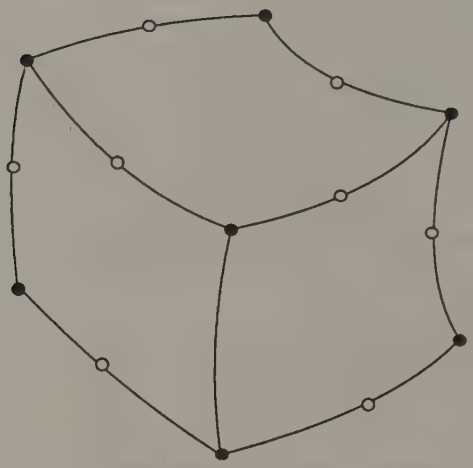


**Figure 4.5** A quadratic cell.

*Validity*: The validity of a cell decomposition is hard to establish. While exhaustive enumeration and octree representation have structural properties that assure validity, a general cell decomposition is just an unordered set of cells. To check its validity, an intersection test for *every* pair of cells is necessary.

*Unambiguity and uniqueness*: A valid cell decomposition completely determines a solid. Cell decompositions are not unique.

*Description languages*: In general, it is very hard to create cell decompositions of interesting objects by direct mechanisms. Usually cell decompositions are created by conversion from other, more convenient representations.

*Conciseness*: Cell decompositions are relatively concise. The usual representation consists of a list of *nodes*, each being a named three-dimensional point, followed by a list of *elements*, each having a type code and a list of point numbers, whose interpretation is determined by the type code.

*Closure of operations*: Usually cell decompositions are used only as input to analysis algorithms that passively examine the model without modifying it. Nontrivial closed algorithms for cell decompositions (such as Boolean set operations) are either unknown or computationally unattractive.

*Computational ease and applicability*: Cell decompositions are indispensable as the input representation for computational algorithms. Finite

element models (FEM) give a prime example of this.

The role of cell decompositions is clear from the above: they are auxiliary representations used for specific computations rather than as general independent solid models in their own right. We shall return to this later in Section 5.2.

## PROBLEMS

4.1. Show that 7/8 of the nodes of an octree are leaves.

*Hint:* Use induction on the depth of the tree.

4.2. How would you classify an octree node against straight cylinder primitive?

4.3. Describe an algorithm for rotating an octree by a integer multiple of 90 degrees.

4.4. Describe algorithms for calculating (a) the set union and (b) the set difference of two octrees.

4.5. Describe an algorithm that can calculate Boolean set operations of two linear quadtrees represented by means of the DF-representation.

4.6. Design a data structure that can be used to represent a cell decomposition consisting of quadratic cells such as the one shown in Figure 4.5.

## BIBLIOGRAPHIC NOTES

Octrees seem to have been invented independently by several workers. The basic works are by Hunter [59], Jackings and Tanimoto [60], Meagher [83,82], and Reddy and Rubin [93].

The major source of information on quadtrees and related data structures is the comprehensive survey [106] of Samet that also includes a large bibliography.

The survey of Shirari [112] offers lots of information on various representations of three-dimensional digital images primarirly from the point of view of image processing applications.

While accurate octree models of curved solids would be very large, it seems possible to use octrees for cases where somewhat rough models are appropriate. In addition to graphics, Yamaguchi *et al.* [135] describe the use of octree models for rough cutting of mechanical parts.

Cell decompositions are the primary data representation for the Finite Element Method. Standard references to FEM are [12,54].

FEM systems typically include a preprocessor that aids the generation of proper FEM meshes. The more powerful preprocessors are actually solid modelers of their own right that can be useful even outside the FEM context proper; for instance, see [23].

Figure 4.1, page 60 is by A. H. J. Christessen and has been previously published as the inside cover figure of *Computer Graphics*, Volume 14, Number 3. ©1980, Association for Computing Machinery, Inc. Reprinted by permission.

Figure 4.4, page 73 has been previously published as Figure 3 of the article "Interactive graphical CAD in mechanical engineering design" by W. S. Elliott in *Computer-Aided Design*, Volume 10, Number 2. ©1978, Butterworth & Co (Publishers) Ltd. Reprinted by permission.

# Chapter 5

# CONSTRUCTIVE MODELS

Decomposition models discussed in the preceding chapter represent solids as a collection of basic elements, combined with a "gluing" operation. In contrast, the *constructive models* to be discussed in this chapter use much more powerful combination operations.

## 5.1 HALF-SPACE MODELS

All constructive models consider solids as point sets of $E^3$. Their basic idea is to start from some sufficiently simple point sets that can be represented directly, and model other point sets in terms of very general combinations of the simple sets. So-called *half-space* models apply this approach in a direct fashion.

### 5.1.1 Half-Spaces

Every point set $A$ can be thought of as having a *characteristic function* $g_A(X) : X \rightarrow \{0, 1\}$ which tells whether a point $X$ is considered to be a member of $A$ or not. In other words, the function $g_A$ must satisfy

$$g_A(X) = 1 \Rightarrow X \in A$$

$$g_A(X) = 0 \Rightarrow X \notin A$$

For very general point sets characteristic functions do not offer much help, because their representation would be as hard as the representation

of the sets themselves. However, for an interesting class of point sets $g_A$ can be represented in terms of a real-valued analytic function $f$ of $x$, $y$, and $z$ defined everywhere in $E^3$. The restriction to analytic functions excludes certain "pathological" objects [94] as explained in Chapter 3. All points $X = (x\ y\ z)$ such that $f(X) \geq 0$ are considered to belong to the point set, while $f(X) < 0$ defines its complement.

Because $f(X) = 0$ divides the whole space into two subsets, point sets defined by $f(X) \geq 0$ and $f(X) \leq 0$ are referred to as *half-spaces*. For instance, functions

$$ax + by + cz + d \geq 0$$

$$x^2 + y^2 - r^2 \leq 0$$

define useful point sets, namely the *planar half-space* that consists of all points on or in the positive side of the plane $ax + by + cz + d = 0$, and the *cylindrical half-space* that consists of all points or or inside an infinite cylinder whose axis = $z$-axis and radius = $r$.

Other half-spaces of interest include the remaining *natural quadratic surfaces* such as spheres and cones, and certain higher-order surfaces such as tori. Observe that this collection includes both unbounded half-spaces (such as the infinite cylinder above) and bounded half-spaces (such as the sphere $x^2 + y^2 + z^2 - r^2 \leq 0$).

Not all surfaces of interest are half-spaces. For instance, the various *surface patches* widely used in surface models are not half-spaces, because they do not divide the space into distinct subsets. This means that the modeling space of a half-space modeler cannot easily be extended to cover also objects bounded by surface patches.

### 5.1.2 Boolean Set Operations

Half-spaces form the basic modeling primitives of half-space models. As half-spaces are point sets, the natural modeling procedures for half-space models are the *Boolean set operations* union ($\cup$), intersection ($\cap$) and set difference ($\setminus$).

Hence, half-space models are constructed by combining instances of half-spaces with Boolean set operations. For instance, to describe a finite cylinder $C$ of length $h$, we need one cylindrical half-space and two planar half-spaces, combined together with the set operation "$\cap$":

$$\begin{aligned} H_1 &: \quad x^2 + y^2 - r^2 \leq 0 \\ H_2 &: \quad z \geq 0 \\ H_3 &: \quad z - h \leq 0 \end{aligned}$$

$$C = \quad H_1 \cap H_2 \cap H_3$$

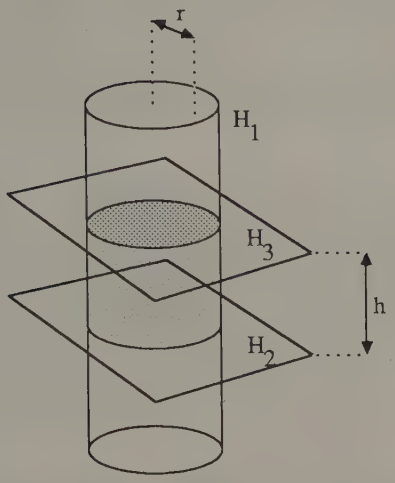



**Figure 5.1** Half-space model of a finite cylinder.

This construction is illustrated in Figure 5.1.

Hence, the modeling space $M$ of a half-space modeler is the class of *Boolean combinations* of the available half-spaces.

### 5.1.3 Representation of Half-Space Models

According to the above, the representation of a half-space model breaks into two parts:

1. *Representation of half-spaces*: The most common approach is to represent half-spaces as instances of half-space types, described with a half-spaces type code and a list of size parameters in some convenient coordinate system, and a transformation matrix that gives the actual location and orientation of the half-space.

   As an alternative to this approach, all quadratic half-spaces can be brought into the same form by writing their defining function $f(X) = f(x, y, z)$ as

$$f(x,y,z) = [x\ y\ z\ 1] \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and storing the 16 coefficients $A_{ij}$, or the 10 coefficients derived by evaluating the matrix products. While this general form offers advantages, e.g., in ray casting (to be explained in Section 5.2.2 on page 92), the surface type coding is more commonly used because of the conveniences it offers, e.g., in the calculation of surface-surface intersections.

2. *Representation of Boolean combinations of half-spaces*: One possibility is offered by the fact that every Boolean expression may be brought into the "sum-of-products" form. TIPS [91], a solid modeler based on the pure half-space approach, follows this approach by representing a solid as the union of the intersections of the individual half-spaces. Hence, a solid S is expressed in TIPS as

$$S = \bigcup_i \bigcap_j H_{ij},$$

where $H_{ij}$ denote the individual half-spaces of $S$. Other approaches to representing combinations of half-spaces are discussed later in this chapter.

### 5.1.4 Properties of Half-Space Models

Properties of pure half-space modelers can be described in the already familiar framework.

*Expressive power*: The modeling space of a half-space modeler is determined by the selection of half-spaces available, and of the generality of the operations available for combining them. Typically, modelers of this class include planar and quadratic half-spaces (such as spherical, cylindrical, and conical surfaces), sometimes even tori. Note, however, that patch surfaces are *not* half-spaces and cannot hence be included into the modeling space of half-space modelers.

*Validity*: Half-spaces are (usually) infinite point sets. Even a combination of them may be infinite, and hence not all combinations are valid solids (if finiteness is a required characteristic of a "solid" point set). TIPS circumvents this difficulty by enforcing the definition of a "box of interest" as a part of each solid description; only the part of space within the box is considered to form the solid.

*Unambiguity and uniqueness*: Each valid combination of half-spaces determines a solid. Hence half-space models are unambiguous. Half-space representations are not unique.

*Description languages*: Instantiation and combination of half-spaces leads easily to text-oriented, relatively simple solid descriptions. With some effort, it is also possible to create a drafting-type graphical user interface.

*Conciseness*: Half-space models are relatively concise: a few hundred half-spaces are usually sufficient to model realistic parts adequately (within the limitations of the modeling space).

*Closure of operations*: Any combination of two half-space models with a Boolean set operation defines a new valid model; hence these operations are closed.

*Computational ease and applicability*: The natural algorithms for half-space modeling are based on so-called *set membership classification*. In particular, the *ray casting approach* to be explained in the context of CSG models in Section 5.2.2 is a simple and general way of attacking the computational problems of half-space models. These families of algorithms are discussed in the next section in connection with the related CSG models.

## 5.2 CONSTRUCTIVE SOLID GEOMETRY

Pure half-space models offer a mathematically rigorous, easily understandable approach to solid modeling. For human users, however, it is easier to operate with bounded primitives instead of the unbounded half-spaces. Observe also that combinations of half-spaces may be unbounded point sets that do not quite match our notion of a valid solid. To avoid the generation of unbounded sets, the so-called *Constructive Solid Geometry* (CSG) approach to solid modeling uses only bounded point sets as its primitives.

### 5.2.1 Representation of CSG Models

CSG adopts the "building block" approach to solid modeling in its pure form. The user of a CSG modeler operates only on parameterized instances of *solid primitives* and Boolean set operations on them. Each primitive, in turn, is defined as a combination of half-spaces; the user has no direct access to individual half-spaces, however. For instance, the user may create planar and cylindrical half-spaces by means of a cylinder primitive, but he cannot directly manipulate them. Hence the mathematical modeling space of CSG models is exactly the same as that of pure half-space models, except that only finite point sets are included.

The most natural way to represent a CSG model is the so-called *CSG tree* that can be defined as follows:

$$
\begin{aligned}
\langle \textit{CSG tree} \rangle \quad ::= \quad & \langle \textit{primitive} \rangle \mid \\
& \langle \textit{CSG tree} \rangle \ \langle \textit{set operation} \rangle \ \langle \textit{CSG tree} \rangle \mid \\
& \langle \textit{CSG tree} \rangle \ \langle \textit{rigid motion} \rangle
\end{aligned}
$$

In the above, <*primitive*> is an instance of a solid primitive, represented through a primitive type identifier and a sequence of dimension

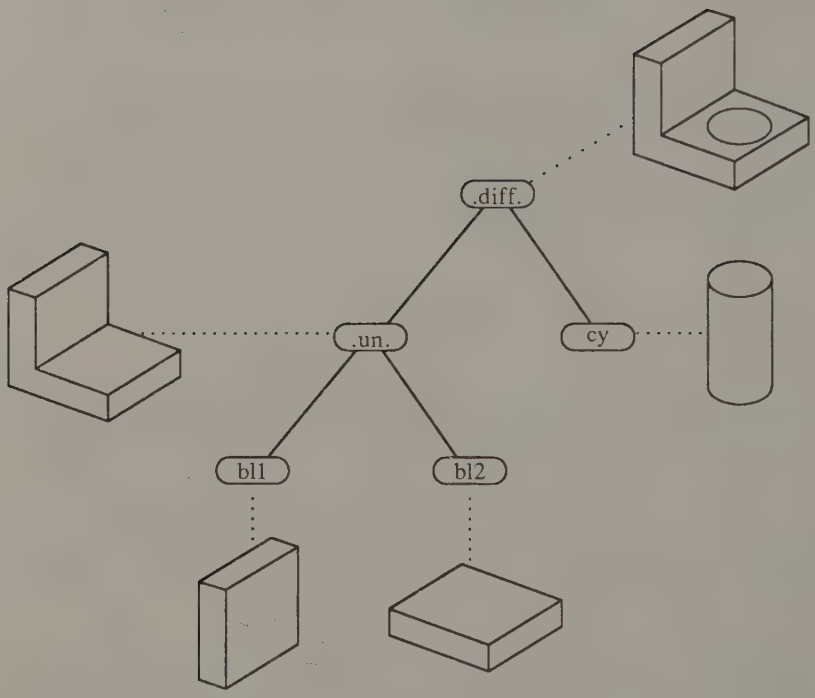



**Figure 5.2** A CSG tree.

parameters. <*rigid motion*> is either a translation or a rotation, and <*set operation*> is one of $\cup$, $\cap$, and $\backslash$.

Hence, primitives are represented in the leaves of the CSG tree, while interior nodes are marked with either a Boolean set operation or with a rigid motion (see Figure 5.2). In this fashion, set operations and motions are interpreted to operate on CSG trees. This naturally leads to the creation CSG trees that model subassemblies of a design, which after appropriate transformations are used several times in the CSG tree representing the assembled design. In this case the binary tree actually becomes a *directed acyclic graph.*

Each primitive is chosen so as to define a bounded point set of $E^3$.

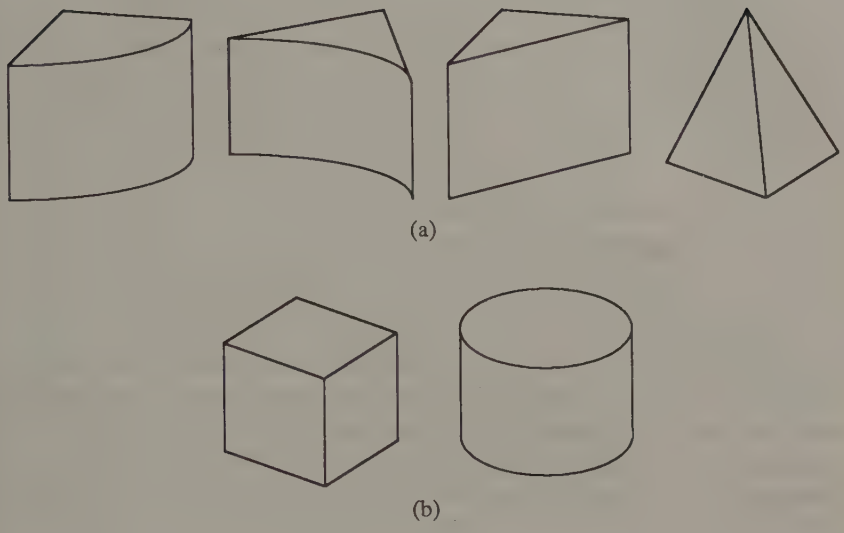



**Figure 5.3** Two collections of CSG primitives.

As the set operations available cannot destroy boundedness, CSG models are guaranteed to define bounded sets. The ease of use of a CSG modeler depends much on the collection of primitives available. For instance, Figure 5.3 depicts two collections of CSG primitives. Note how collection (a) includes various wedges to aid the rounding of edges often needed in mechanical parts. Observe that all primitives of the figure can be expressed as a Boolean combination of simple half-spaces (in this case, planes and cylinders).

The actual domain of a CSG modeler depends on the variety of half-spaces available in its primitives, on the available rigid motions, and on the available set operations. Note that the two collections in Figure 5.3 have the same domain despite different primitives, because the underlying collection of half-spaces available is the same in both cases. (Actually, a CSG modeler with just one cylinder primitive has exactly the same domain.)

### Regular Set Operations

Some combinations of CSG primitives (or half-spaces) do not quite satisfy our notion of "solidity." Consider, for instance, the case depicted in Figure 5.4. According to the ordinary definition of point set operations,

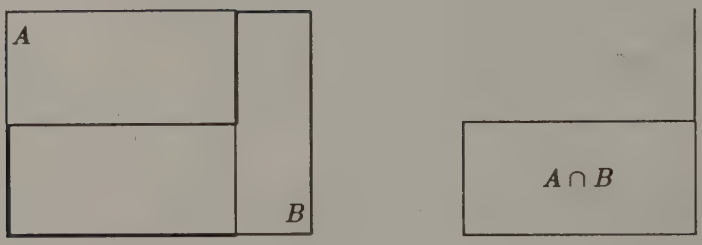



**Figure 5.4** A nonregular set operation.

the intersection of the two objects consists of a rectangular object plus a "dangling" line segment (b). This effect is undesirable, for instance, if set operations are used to test the validity of an assembly by intersecting all component pairs and checking whether the result is an empty object. In that case, we would prefer that the intersection of merely touching components is empty.

In light of the point-set characterization of solidity, the problem of the resulting object is that it is not regular: the "dangling line" violates the Definition 3.3 of regularity on page 33.

The concept of regularity is introduced into CSG by considering its set operations to be the regularity-preserving variants of the usual point-set operations as defined below.

**Definition 5.1** *The* regularized set operations *union*, intersection*, and set difference*, denoted by* $\cup^*$, $\cap^*$ *and* $\setminus^*$ *are defined as*

$$\begin{aligned} A \cup^* B &= c(i(A \cup B)) \\ A \cap^* B &= c(i(A \cap B)) \\ A \setminus^* B &= c(i(A \setminus B)) \end{aligned}$$

*where* $\cup$, $\cap$, *and* $\setminus$ *denote the usual set operations.*

If CSG primitives are chosen to be bounded regular sets, regularized set operations have the desirable property of being *algebraically closed* in the class of bounded regular sets. That is, every CSG tree is guaranteed to define a bounded regular set.

### 5.2.2 Algorithms for CSG Models

The CSG tree can be viewed as an implicit description of the geometry of the solid modeled that must be "evaluated" in some fashion in order to create graphical output or perform calculations.

The very structure of the CSG tree suggests the use of the very powerful and general "divide-and-conquer" approach to the design of algorithms for CSG trees. The basic idea of divide and conquer is to divide the problem in some fashion into two parts, recursively solve each part, and join the partial solutions to get the total solution. The recursion terminates when the problem has been subdivided into its "primitive" parts that allow a direct solution.

When applied to CSG trees, the natural way to subdivide a problem is to process the two subtrees of each interior node denoting a Boolean set operation of the tree separately. The recursion ends at leaves of the tree, where the problem is solved for one primitive. Solutions of subproblems are combined while taking the set operation at the node into respect.

Divide and conquer leads to efficient algorithms if the combination of subsolutions is simple and the problem can always be split into parts of approximately same size. Unfortunately, CSG trees are usually far from being balanced. Program 5.1 gives a generic divide and conquer algorithm for processing CSG trees. The following sections give specific instances of this schema.

### Set Membership Classification

So-called *set membership classification* [121] algorithms are a particularly useful class of algorithms based on the divide and conquer approach. In general, a set membership classification algorithm works on two point sets, namely the *candidate set* $C$ and the *reference set* $R$. The algorithm is expected to *classify* $C$ against $R$ by forming three sets $CinR$, $ConR$, and $CoutR$ representing the parts of $C$ inside, on the boundary, and outside of $R$.

Specification of the "meaning" and the representations of the sets involved ($C$, $R$, $CinR$, $ConR$, $CoutR$) defines a particular set membership classification algorithm. For instance, the algorithm for classifying a finite line segment ("edge") against a CSG tree is cast into the general schema by choosing the sets as follows:

- $C$: An edge $E$ given in terms of an ordered pair of coordinate triples;
- $R$: A CSG tree $S$;
- $CinR$, $ConR$, $CoutR$: Sets $EinS$, $EonS$, and $EoutS$, each being a set of nonempty subsegments of $E$ such that $EinS \cup EonS \cup EoutS = E$, and their elements are inside, on the boundary, or outside $S$, respectively.

```
/* evaluate property P of a CSG tree */
P *Tree_P(S, args)
CSG_Tree *S;
{
        if(S->op == <primitive>)
                return(Primitive_P(S, args));
        else    return(Combine_P(Tree_P(S->Left, args),
                                 Tree_P(S->Right, args),
                                 S->Op));
}

/* evaluate P for a primitive */
P *Primitive_P(S, args)
{
        ...
}

/* combine two evaluations of P with set operation Op */
P *Combine_P(Left_P, Right_P, Op)
{
        ...
}
```

**Program 5.1** Divide and conquer for CSG trees.

```
/* set membership classification of an edge vs. a CSG tree */
M *Tree_M(S, E)
CSG_Tree *S;
Edge *E;
{
        if(S->Op == <primitive>)
                return(Prim_M(S, E));
        else    return(Combine_M(Tree_M(S->Left, E),
                                 Tree_M(S->Right, E),
                                 S->Op));
}

/* classify E against a primitive */
M *Prim_M(S, E)
{
        ...
}

/* combine two classifications of E */
M *Combine_M(Left_M, Right_M, Op)
{
        ...
}
```

**Program 5.2** Edge-solid classification.

The resulting algorithm of Program 5.2 nicely fits in the general approach of Program 5.1. The "property" evaluated is now the triple $M = \langle EinS, EonS, EoutS \rangle$. All we need to supply is an algorithm for classifying an edge against a primitive (`Prim_M` in Program 5.2), and an algorithm for combining two classifications (`Combine_M`).

**Classification Against Primitive** To do its task properly, the procedure `Prim_M` must calculate the intersections (if any) between $E$ and the primitive, and subdivide $E$ accordingly. For instance, in the case of Figure 5.5 the desired result of the primitive classification is the triple

$$\begin{aligned} M = <&\{(p_2, p_3)\}, && (EinS) \\ &\{\}, && (EonS) \\ &\{(p_1, p_2), (p_3, p_4)\} >. && (EoutS) \end{aligned}$$

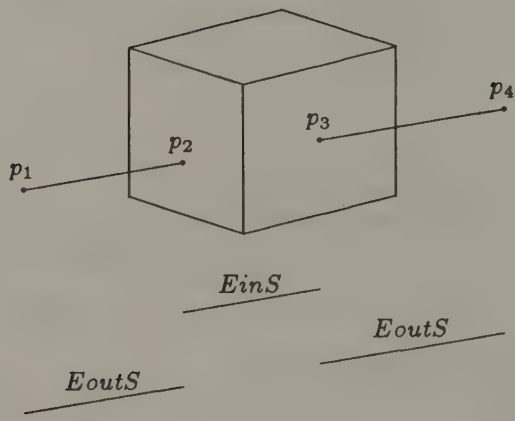



**Figure 5.5** Edge vs. block primitive.

As all CSG primitives actually are combinations of half-spaces, the basic task needed is the classification of $E$ against a half-space. This is a relatively straightforward operation.

For instance, suppose that we need to classify an edge $E$ against a cylinder half-space. By transforming $E$ appropriately, we can arrange things so that the cylinder is of the form

$$x^2 + y^2 - r^2 = 0,$$

and that $E$ is represented by the parametric equation

$$E(t) = p_1 + t(p_2 - p_1),\ t = [0, 1]$$

where $p_i = (x_i\ y_i\ z_i)$ are the end points of $E$, and $t$ is the line parameter. This gives us three simultaneous equations

$$\begin{aligned} x^2 + y^2 &= r^2 \\ x &= x_1 + t(x_2 - x_1) \\ y &= y_1 + t(y_2 - y_1). \end{aligned}$$

By substituting the second and third equation into the first, we get a second-order equation in $t$. If the equation has no real solutions, $E$ does not intersect the cylinder at all; if a double root occurs, $E$ is tangent to the cylinder; otherwise, the parameter values of two intersection points are given. What remains is to check whether the intersections occur within the

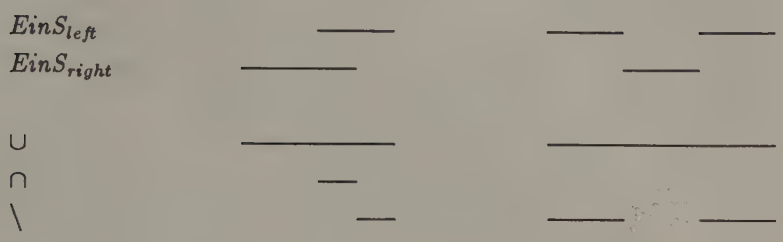



**Figure 5.6** Combination rules.

edge, i.e., whether the parameter values are within 0 and 1. In the most complicated case of tori, a similar procedure yields a fourth-order polynomial in $t$, i.e., still something that can be solved with standard techniques.

**Combination of Classifications** Each of the results of edge-primitive classification ($EinS$, $EonS$, $EoutS$) consists of a sequence of edge segments. By arranging these sequences so that the segments appear in sorted order along $E$, the combination of classifications can be reduced to the merging of sets of line segments. The primitive task of this operation is the combination of just two edge segments by the Boolean set operation $Op$. For $EinS$ and $EoutS$ this leads to the combination rules illustrated in Figure 5.6.

Unfortunately, the rules for combining segments of the "left" $EonS$ with the "right" $EonS$ are not quite so simple. As illustrated in Figure 5.7, the combination rules for this "on-on"-case must take also the orientations of the respective surfaces into account.

**Wire Frame Generation** As an example on the use of the edge-solid classification, let us consider the *wire frame problem*:

> *Given a CSG tree $S$, determine a set* WireFrame *of line segments that form the "wire-frame" figure of $S$.*

Based on the edge-solid classification, the wire frame generation algorithm can be combined from the following steps:

1. *Generate*: Generate the set of all "tentative" edges by calculating the set of pairwise intersection curves of all half-spaces appearing in $S$.

2. *Classify*: Classify each tentative edge against $S$, and append the component $EonS$ to the result.

```
WireFrameGen(S)
CSGTree S;
{
    /* Generation step: */
    Tentative_list = empty;
    for each half-space G of S
    {
        for each half-space H of S
        {
            Tentative_list = union(Tentative_list, Intersect(G, H));
        }
    }

    /* Classification step: */
    WireFrame = empty;
    for each edge E of Tentative_list
    {
        <EinS, EonS, EoutS> = Tree_M(S, E);
        WireFrame = union(WireFrame, EonS);
    }
}
```

**Program 5.3** Wire frame generation algorithm.

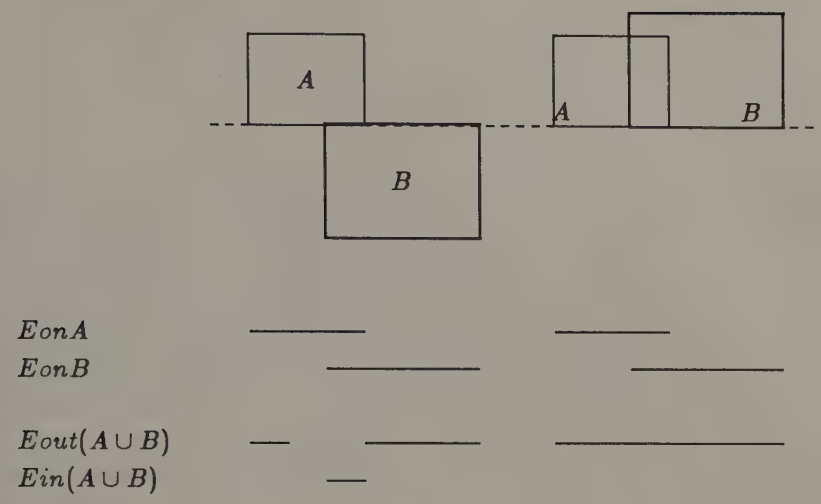



**Figure 5.7** Combination rules for "on-on"-cases.

Program 5.3 outlines the resulting algorithm. The use of the complete three-way edge-solid classification may of course appear an overkill here, as only the component *EonS* really is needed. For more practical implementations of the algorithm, the reader is referred to Tilove [119].

**Boundary Evaluation** The generation of images with hidden lines removed requires a much more complicated algorithm that calculates the "faces" of the solid modeled via the CSG tree. This process is termed the *boundary evaluation.* Ordinarily, boundary evaluation is based on set membership classification between "primitive faces" (faces derived from CSG primitives) and CSG trees [96]. Of course, the capability of calculating intersections of quadratic surfaces is required [109].

In CSG modelers PADL-1 [126,97], PADL-2 [21], and GMSOLID [17] boundary evaluation is used to construct a complete boundary model based in the CSG tree. In particular, PADL-2 pursues so-called "incremental boundary evaluation" [119] that updates the boundary model so as to reflect changes in the CSG tree. These and other CSG modelers that can construct a boundary model from a CSG model can, of course, utilize algorithms for boundary models whenever they seem more appropriate than the self-contained methods for CSG models. Boundary models and their algorithms are discussed in Chapter 6.

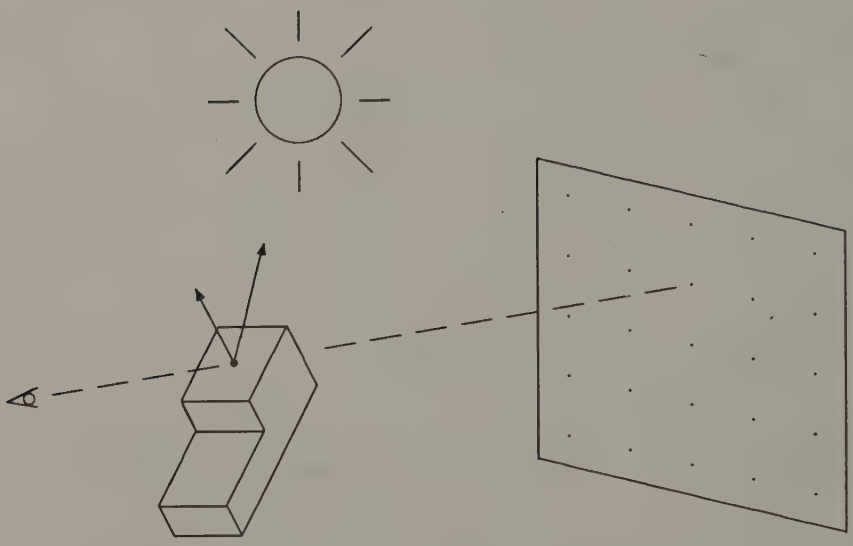


**Figure 5.8** The ray casting paradigm.

### Ray Casting

Another general approach to algorithm design for CSG models is the *ray casting* approach [104]. Ray casting is by its very nature approximative: instead of "evaluating" a CSG tree exactly, the tree is "sampled" along a finite number of lines, hence reducing the calculation of three-dimensional set operations to that of one-dimensional ones.

The term arises from the problem of generating an image of a CSG model for a color raster scan terminal (see Figure 5.8). In the ray casting paradigm, a "view ray" is sent from the location of the viewer through each pixel of the image. Based on the location where the ray first hits the object viewed, the shade and the intensity of the pixel are calculated. By sending secondary rays from the hit location, special visual effects such as shadowing, transparency, and mirroring can be created. This process is termed *ray tracing.*

The ray casting approach is actually a variation of the edge-solid classification, where instead of a finite edge, a semi-infinite ray is classified by a *ray classification* procedure. As a two-way "in-out" classification is sufficient, and as it can be assumed that the location of the "eye" is outside the solid, the ray classification procedure can represent the results by

means of a sequence of intersection points between the ray and the solid. An adequate representation is to have two arrays

$$\begin{array}{cccccc} t(1) & t(2) & t(3) & t(4) & \ldots \\ s(1) & s(2) & s(3) & s(4) & \ldots \end{array}$$

where $t(i)$'s give the parameter values of intersection points in a sorted order, and $s(i)$'s identify the half-spaces intersected at these points. The ray segments $[0, t(1)]$, $[t(2), t(3)]$, and so on are considered to occur outside the solid, and segments $[t(1), t(2)]$, $[t(3), t(4)]$ inside of it.

The implementation of the ray-classification procedure follows similar lines as the edge-solid classification, being only simpler. For clarity, the outline of the algorithm given in Program 5.4 indicates the intersection tests needed between the ray and the half-spaces of the primitives.

The image generation algorithm can now simply pick the first intersection of the sequence, and calculate the color based on the information it has on normal vector and color of the surface intersected and the locations of light sources.

Ray casting is useful not only for color image generation, but also for calculating line drawings and integral properties of solids; see, e.g., [44]. The problem is that the amount of computation needed is large: to calculate an image at $1000^2$ resolution, one million ray classifications are needed. As each classification is a recursive algorithm potentially involving numerical solutions of fourth order polynomials, the speed of ray casting algorithms is very far from interactive.

### Integral Properties

The approximative evaluation of integral properties (such as volume) of CSG models is ordinarily based on conversion algorithms that (implicitly) construct a decomposition model approximating the solid. The evaluation of the property then reduces to calculating the contribution of each cell of the decomposition to the total result.

The conversion algorithms are generally based on the set membership classification and ray casting approaches. Several alternative arrangements of cells are possible, each corresponding to a particular set membership classification algorithm. Various alternatives are shown in Figure 5.9 [98,69,68].

For instance, the case of "column decomposition" corresponds with ordinary ray casting. A column is determined by sending a ray along its center; where the ray is inside the solid, a solid column is created. This algorithm is used in solid modelers employing the ray casting approach, e.g., in the SYNTHAVISION$^{TM}$ solid modeler [45].

```
Classification
RayCast(S, R)
CSGTree *S;
Ray *R;
{
    if(S->Op == <set operation>)
    {
        LeftClassification = RayCast(S->Left, R);
        RightClassification = RayCast(S->Right, R);
        return(Combine(LeftClassification,
                       RightClassification,
                       S->Op);
    }
    else
    {
        switch(S->Op)
        {
            case "block":
                do 6 ray-plane intersection tests
            case "sphere":
                do 1 ray-quadratic intersection test
            case "cylinder":
                do 2 ray-plane and 1 ray-quadratic tests
            case "cone":
                do 1 ray-plane and 1 ray-quadratic test
            case "torus":
                do 1 ray-quartic intersection test
        }
        Classification = results of tests;
        return(Classification);
    }
}
```

**Program 5.4** Ray classification algorithm.

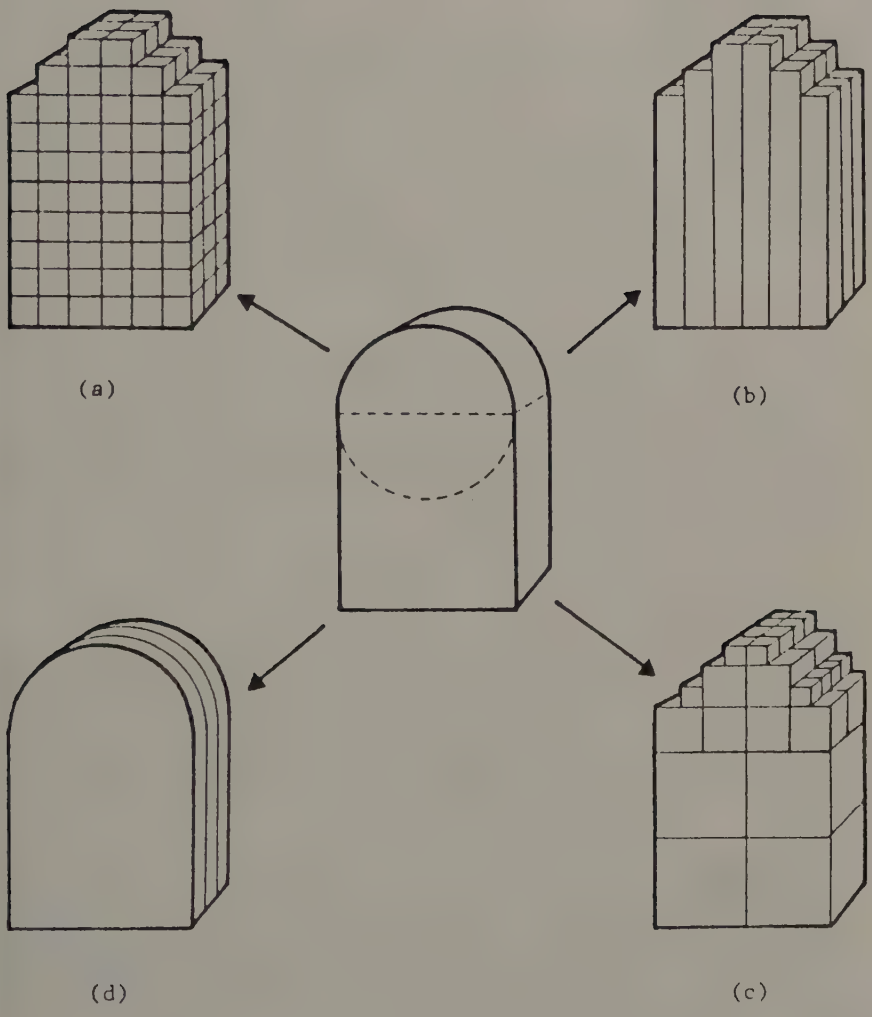



**Figure 5.9** CSG-to-cell conversions.

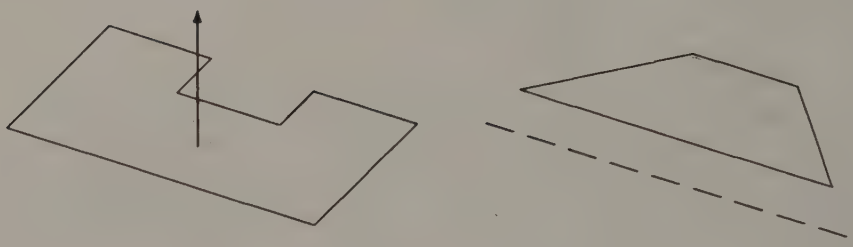


**Figure 5.10** Sweeping primitives.

The simplest case, exhaustive enumeration, is based on *point-solid classification*: given a point $P$ (the center point of the cell), determine whether P is inside, on the boundary, or outside the solid. CSG is directly useful for answering this kind of a question as classification with respect to primitives amounts to substituting $P$ to the defining function of each half-space. For the two-way classification, the combination rules are simple and left to the reader. This method is used in TIPS.

### 5.2.3 Generalized CSG Primitives

To apply the set membership classification and ray casting approaches to CSG algorithms for a particular CSG model, the essential task reduces to writing procedures that perform the underlying operations of these algorithms. Correspondingly, *any* primitive for which such procedures can be written can potentially be included into a CSG modeler. In particular, there is no need to restrict CSG primitives to be simple collections of half-spaces.

Many CSG-based systems follow this approach, and allow the inclusion of *generalized CSG primitives* of various types into their CSG models. For instance, *polyhedral primitives* can be added into a ray casting modeler by offering sufficient facilities for their creation, and a ray casting algorithm for arbitrary polyhedra. This approach leads to a hybrid of the CSG and boundary modeling approaches.

Another potentially useful addition is the inclusion of *sweeping primitives*, modeling primitives of the *sweeping model* approach to solid modeling. A *translational sweeping primitive* is defined by a *base set*, a bounded subset of a plane, and a *sweeping direction*; informally, the primitive consists of the set of points "swept" by the base set as it moves along the sweeping direction. A *rotational sweeping primitive* is similarly defined by a base set and a rotation axis, and consists of the set of points swept by the

base set as it rotates around the axis. Of course, restrictions on admissible base sets must be posed so as to guarantee that the resulting objects are valid solids.

In this context, the essential fact is that there are algorithms, for example, for the ray casting of translational and rotational sweeping primitives directly (without conversion to some other representation); see [62,125,124]. Hence they can be included into the domain of a ray casting modeler. Moreover, "two-and-half-dimensional" models generated by sweeping simple outlines consisting of straight lines and arcs along a normal axis can be converted directly to CSG models; see [128].

### 5.2.4 Properties of CSG Models

The properties of CSG models can be summarized similarly as those of other models we have discussed.

*Expressive power*: Depends on the class of half-spaces available. Cannot easily be extended to cover surface patches.

*Validity*: Every CSG tree is guaranteed to model a valid solid object, provided that the primitives are valid (i.e., bounded regular sets).

*Unambiguity and uniqueness*: Every CSG tree unambiguously models a solid. They are not unique.

*Description languages*: Usually textual languages. It is possible to include a graphical interface into a CSG modeler.

*Conciseness*: CSG trees are in principle relatively concise. In practical modelers, they tend to grow as other information aiming at efficient graphical operations is attached to the basic CSG tree.

*Closure of operations*: Set operations are algebraically closed for CSG trees.

*Computational ease and applicability*: The computational power some important CSG algorithms (such as boundary evaluation) is poor. However, as their basic steps are very simple, they are prime candidates for VLSI implementation. In cases such as the point classification for conversion to a decomposition model, the divide and conquer approach generally leads to efficient algorithms, although their efficiency deteriorates if the CSG tree is ill-balanced.

CSG models are based on rigorous mathematical background, and the families of algorithms available for them are well understood and applicable to many problems. Because of these reasons, many commercial CSG modelers are available and in wide use.

## PROBLEMS

5.1. Give a Boolean expression of half-spaces for each of the CSG primitives of Figure 5.3.

5.2. Specify and write a ray-sphere intersection procedure based on the general outline of Program 5.4. In addition to the ray parameter value of the intersection point, the procedure should also calculate the surface normal of the sphere at the intersection point for use of the ray tracing program of Problem 5.3.

5.3. Based on the ray-sphere intersection procedure of Problem 5.2, specify and write a ray tracing program that can display images (possibly intersecting) spheres on a color raster display.

*Hint:* You will need to apply a suitable shading method based on the surface normal of the intersection of the ray with a sphere and the location(s) of the light source(s). See, e.g., [39,88] for additional information.

5.4. Generalize your ray tracing program of Problem 5.3 by including additional primitive types, such as blocks, cylinders, and cones. Try not be become too hooked on pretty pictures!

## BIBLIOGRAPHIC NOTES

The basic ideas of constructive modeling can be traced back to late 60's and early 70's. Ray casting was probably introduced by Appel [5]. In this context, many aspects of CSG modeling were quite early developed. Probably the first practical modeler was the original SYNTHAVISION system [46].

As a general technique for CSG modeling, ray casting was introduced by Roth [104].

The rigorous theory of CSG modeling was developed by the workers of the PADL-project at the University of Rochester [126,97,94]. The project constructed two successive CSG modelers, PADL-1 and PADL-2, which became widely known in the industry and the academia. They were eventually used as a basis for commercial development.

Advanced algorithmic techniques for CSG diminish the amount of processing by using several techniques. The thesis of Tilove [119] is a basic reference to many of them. A good example of these techniques is the *null object detection* algorithm described in [120]. A null object detection algorithm is expected to test whether a given CSG tree models the empty set or not.

The *pruning* of a CSG tree tries to diminish the amount of processing by modifying a CSG tree into a smaller equivalent tree. Woodwark uses a spatial subdivision and pruning technique to speed up the generation of images from a CSG representation [133,132]. Samet and Tamminen [107] use a similar technique for the conversion of CSG models to bintrees.

Figure 5.9, page 95 has been previously published as Figure 1 of the article "Algorithms for computing the volume and other integral properties of solids" by Y. T. Lee and A. A. G. Requicha in *Communications of the ACM*, Volume 25, Number 9. ©1982, Association for Computing Machinery, Inc. Reprinted by permission.

# Chapter 6

# BOUNDARY MODELS

Decomposition models and constructive models both view solids as point sets, and seek representations for the point set either by discretizing it or by constructing it from simpler point sets. In contrast to these models, *boundary models* represent a solid indirectly through a representation of its bounding surface.

## 6.1 BASIC CONCEPTS

Historically, boundary models emerged from the polyhedral models used in computer graphics for representing objects and scenes for hidden line and surface removal. They can be viewed as "enhanced" graphical models that attempt to overcome the problems of graphical models by including a complete description of the bounding surfaces of the object.

Boundary models are based on the surface-oriented view to solid modeling represented in Section 3.5. That is, they represent a solid object by dividing its surface into a collection of *faces* in some convenient fashion. Usually, the division is performed so that the shape of each face has a compact mathematical representation, e.g., that the face lies on a single planar, quadratic, toroidal, or parametric surface. In order to guarantee that the subdivision corresponds with a legal plane model as discussed in Section 3.5, it is usually required to satisfy certain "topological" criteria to be discussed in the sequel.

The portion of the underlying surface that forms the face is "chalked out" in terms of a closed curve that lie on the surface. A face may well have several bounding curves, provided that they define a connected object. That is, faces are like "continents" that may have "lakes"; "isles" on lakes do not belong to faces, however.

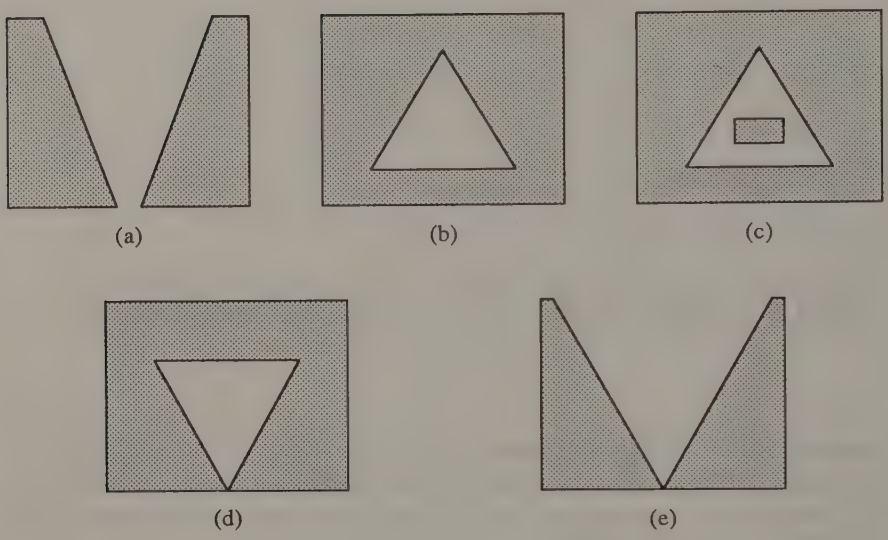



**Figure 6.1** Definition of a face.

The definition of a face is depicted in Figure 6.1. In the figure, cases (a) and (c) are disconnected 2-dimensional sets and would be represented as two faces. Cases (d) and (e) are somewhat exceptional faces whose boundaries "touch" themselves. Usually case (d) would be considered a good face, whereas (e) would be represented as two faces. Observe that the interior of (d) is connected, and of (e) not. Case (c) is a face with two boundaries.

In turn, the bounding curves of faces are represented through a division into *edges*. Analogously to the above, edges are chosen so as to have a convenient representation, say, a parametric equation. The portion of the curve that forms the edge is chalked out in terms of two *vertices*.

Figure 6.2 illustrates the basic components of a boundary model. In the figure, the surface of the object is divided into an enclosing set of faces (a), each of which is represented in terms of its bounding polygon (b), in turn represented in terms of edges and vertices (c).

## 6.2 BOUNDARY DATA STRUCTURES

The three object types *face*, *edge*, and *vertex*, and the geometric information attached to them form the basic constituents of boundary models. In

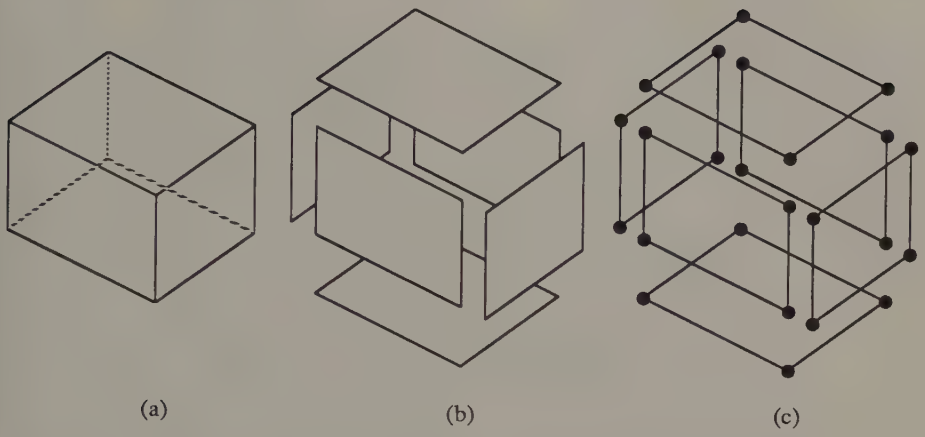



**Figure 6.2** Basic constituents of boundary models.

addition to geometric information such as face and curve equations and vertex coordinates, a boundary model must also represent how the faces, edges, and vertices are related to each other. It is customary to bundle all information of the geometry of the entities under the term *geometry* of a boundary model, and similarly information of their interconnections under the term *topology*.

All boundary models represent faces in terms of explicit nodes of a boundary data structure. After that, many alternatives for representing the geometry and the topology of a boundary model are possible, some of which are described below. For simplicity and clarity, we shall illustrate them in terms of alternative representations for the rectilinear block shown in Figure 6.3. Reference [8] discusses further alternatives and provides information on the data structures used in a number of modelers.

### 6.2.1 Polygon-Based Boundary Models

A boundary model that has only planar faces is called a *polyhedral model*. Because all edges of a polyhedron are straight line segments, it is possible to shrink the boundary data structure considerably in this important special case.

In the simplest variation faces are represented as *polygons*, each polygon consisting of a sequence of coordinate triples. A solid consists of a collection of faces grouped together in terms of, say, a table of face identifiers or a linked list of face nodes. Sometimes even the grouping information is elim-

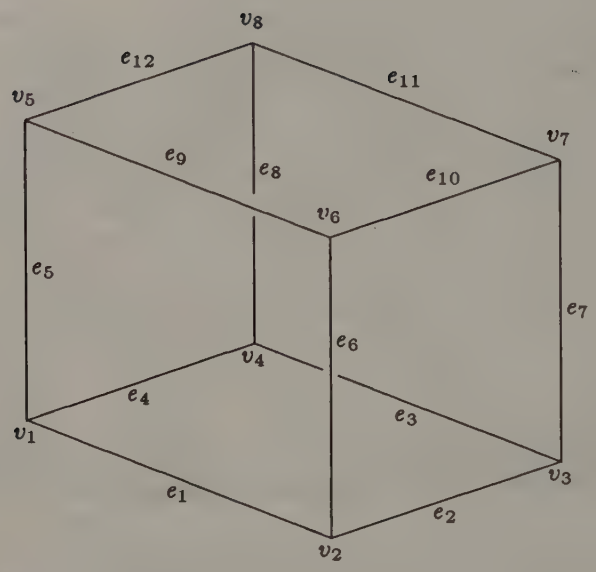



**Figure 6.3** A sample object.

inated and face relationships are completely implicit. This representation is usually used in *graphical metafiles* of graphics systems.

### 6.2.2 Vertex-Based Boundary Models

In a polygon-based boundary model vertex coordinates appear as often as the vertex appears in a face. This redundancy can be eliminated by introducing vertices as independent entities of the boundary data structure. In this case, vertex identifiers (or something equivalent, such as pointers to vertex nodes) are associated with faces. This approach leads us to various *vertex-based boundary models.*

Note that the sample representation of Figure 6.4 lists vertices of each face in a *consistent order*, namely clockwise as seen from the outside of the cube. This consistent orientation is useful in many algorithms. For instance, in hidden line or surface removal, it allows the elimination of *back faces* on the basis of face normal vectors pointing consistently away from the material. In the case of Figure 6.4, faces $f_1$, $f_4$, and $f_5$ would immediately be discarded by the hidden line or surface removal algorithm.

The representation of Figure 6.4 does not include any surface informa-

| *vertex* | *coordinates* |
|---|---|
| $v_1$ | $x_1\ y_1\ z_1$ |
| $v_2$ | $x_2\ y_2\ z_2$ |
| $v_3$ | $x_3\ y_3\ z_3$ |
| $v_4$ | $x_4\ y_4\ z_4$ |
| $v_5$ | $x_5\ y_5\ z_5$ |
| $v_6$ | $x_6\ y_6\ z_6$ |
| $v_7$ | $x_7\ y_7\ z_7$ |
| $v_8$ | $x_8\ y_8\ z_8$ |

| *face* | *vertices* |
|---|---|
| $f_1$ | $v_1\ v_2\ v_3\ v_4$ |
| $f_2$ | $v_6\ v_2\ v_1\ v_5$ |
| $f_3$ | $v_7\ v_3\ v_2\ v_6$ |
| $f_4$ | $v_8\ v_4\ v_3\ v_7$ |
| $f_5$ | $v_5\ v_1\ v_4\ v_8$ |
| $f_6$ | $v_8\ v_7\ v_6\ v_5$ |

**Figure 6.4** A vertex-based boundary model.

tion at all; as all faces are planar, their geometries are completely defined through the coordinates of their vertices. On the other hand, if face equations would be needed often for numerical calculations (say, for shading), they could be associated with faces.

In general, many choices as to what information is stored explicitly and what information is left implicit (i.e., to be computed as needed) must be made when implementing a boundary model. In Figure 6.4, the explicit inclusion of vertex coordinates implicitly gives us also face equations. (The opposite approach of storing face equations explicitly and leaving vertex coordinates implicit leads to a rather peculiar half-space model appropriate for convex objects only.)

If some redundant geometric information is stored (like the face equations in the example above), subtle numerical problems may arise. Suppose that due to small inaccuracies in floating point calculations, vertices of a face do not lie exactly on the plane defined by its face equation. Which information is considered "correct"?

### 6.2.3 Edge-Based Boundary Models

If curved surfaces are present in a boundary model, it becomes useful to include also edge nodes explicitly in the boundary data structure to be able to store information of intersection curves of faces. (Edge nodes are also useful for recording topological relationships as discussed in the next section.)

An *edge-based boundary model* represents a face boundary in terms of a closing sequence of edges, or *loop* for short. Vertices of the face are represented only through edges. This approach leads us to the model of Figure 6.5.

| *edge* | *vertices* |
|---|---|
| $e_1$ | $v_1$ $v_2$ |
| $e_2$ | $v_2$ $v_3$ |
| $e_3$ | $v_3$ $v_4$ |
| $e_4$ | $v_4$ $v_1$ |
| $e_5$ | $v_1$ $v_5$ |
| $e_6$ | $v_2$ $v_6$ |
| $e_7$ | $v_3$ $v_7$ |
| $e_8$ | $v_4$ $v_8$ |
| $e_9$ | $v_5$ $v_6$ |
| $e_{10}$ | $v_6$ $v_7$ |
| $e_{11}$ | $v_7$ $v_8$ |
| $e_{12}$ | $v_8$ $v_5$ |

| *vertex* | *coordinates* |
|---|---|
| $v_1$ | $x_1$ $y_1$ $z_1$ |
| $v_2$ | $x_2$ $y_2$ $z_2$ |
| $v_3$ | $x_3$ $y_3$ $z_3$ |
| $v_4$ | $x_4$ $y_4$ $z_4$ |
| $v_5$ | $x_5$ $y_5$ $z_5$ |
| $v_6$ | $x_6$ $y_6$ $z_6$ |
| $v_7$ | $x_7$ $y_7$ $z_7$ |
| $v_8$ | $x_8$ $y_8$ $z_8$ |

| *face* | *edges* |
|---|---|
| $f_1$ | $e_1$ $e_2$ $e_3$ $e_4$ |
| $f_2$ | $e_9$ $e_6$ $e_1$ $e_5$ |
| $f_3$ | $e_{10}$ $e_7$ $e_2$ $e_6$ |
| $f_4$ | $e_{11}$ $e_8$ $e_3$ $e_7$ |
| $f_5$ | $e_{12}$ $e_5$ $e_4$ $e_8$ |
| $f_6$ | $e_{12}$ $e_{11}$ $e_{10}$ $e_9$ |

**Figure 6.5** An edge-based model.

The data structure indicates an *orientation* for each edge; say, edge $e_1$ is considered to be (positively) oriented from vertex $v_1$ to vertex $v_2$. Again, faces are consistently oriented, i.e., their edges are listed clockwise as viewed from the outside of the block. Note that each edge occurs in exactly two faces, once in its positive orientation, and once in the opposite (negative) orientation.

**Winged-Edge Data Structure** The inclusion of explicit nodes for each of the basic object types (face, edge, and vertex) opens the door for elaborating an edge-based boundary model further. For instance, to aid algorithms such as hidden surface removal and shading, explicit face-face neighborhood information can be added to the data structure of Figure 6.5 by associating edges with the identifiers of the two faces they separate.

The so-called *winged-edge data structure*, first introduced by Baumgart [13,14] gives a primary example of such elaborated edge-based boundary models. It goes one step further by representing also the loop information in edge nodes.

Because each edge $e$ appears in exactly two faces, exactly two other edges $e'$ and $e''$ appear after $e$ in these faces. Moreover, because of the consistent orientation of faces, $e$ occurs exactly once in its positive orientation, and exactly once in the opposite orientation.

The winged-edge data structure takes advantage of these structural properties by associating the identifiers of the two "next" edges with an

| *edge* | *vstart* | *vend* | *ncw* | *nccw* |
|---|---|---|---|---|
| $e_1$ | $v_1$ | $v_2$ | $e_2$ | $e_5$ |
| $e_2$ | $v_2$ | $v_3$ | $e_3$ | $e_6$ |
| $e_3$ | $v_3$ | $v_4$ | $e_4$ | $e_7$ |
| $e_4$ | $v_4$ | $v_1$ | $e_1$ | $e_8$ |
| $e_5$ | $v_1$ | $v_5$ | $e_9$ | $e_4$ |
| $e_6$ | $v_2$ | $v_6$ | $e_{10}$ | $e_1$ |
| $e_7$ | $v_3$ | $v_7$ | $e_{11}$ | $e_2$ |
| $e_8$ | $v_4$ | $v_8$ | $e_{12}$ | $e_3$ |
| $e_9$ | $v_5$ | $v_6$ | $e_6$ | $e_{12}$ |
| $e_{10}$ | $v_6$ | $v_7$ | $e_7$ | $e_9$ |
| $e_{11}$ | $v_7$ | $v_8$ | $e_8$ | $e_{10}$ |
| $e_{12}$ | $v_8$ | $v_5$ | $e_5$ | $e_{11}$ |

| *vertex* | *coordinates* |
|---|---|
| $v_1$ | $x_1\ y_1\ z_1$ |
| $v_2$ | $x_2\ y_2\ z_2$ |
| $v_3$ | $x_3\ y_3\ z_3$ |
| $v_4$ | $x_4\ y_4\ z_4$ |
| $v_5$ | $x_5\ y_5\ z_5$ |
| $v_6$ | $x_6\ y_6\ z_6$ |
| $v_7$ | $x_7\ y_7\ z_7$ |
| $v_8$ | $x_8\ y_8\ z_8$ |

| *face* | *first edge* | *sign* |
|---|---|---|
| $f_1$ | $e_1$ | + |
| $f_2$ | $e_9$ | + |
| $f_3$ | $e_6$ | + |
| $f_4$ | $e_7$ | + |
| $f_5$ | $e_{12}$ | + |
| $f_6$ | $e_9$ | − |

**Figure 6.6** The winged-edge data structure.

edge node. By convention, these data are denoted by *ncw* and *nccw* for "next clockwise" and "next counterclockwise"; in particular, *ncw* identifies the next edge in the face where the edge occurs in its positive orientation, and *nccw* the next edge in the other face.

By virtue of this edge representation, faces only need to include the identifier of an arbitrary edge and a bit that indicates its orientation. Figure 6.6 gives an example of the winged-edge data structure. Signs + and − denote the orientations of the edge.

Starting from the edge directly associated with a loop, all other edges may be retrieved by following the *ncw* and *nccw* pointers. For instance, in the case of Figure 6.6, the boundary of face $f_5$ is extracted by starting from $e_{12}$. The next edge is given by $ncw(e_{12}) = e_5$. Because $e_5$ was traversed in its negative orientation (which can be seen by examining vertex identifiers), the next edge is now given by $nccw(e_5) = e_4$. Similarly we get

$nccw(e_4) = e_8$ as the next edge. It is traversed in its positive orientation, so the next edge is $ncw(e_8) = e_{12}$ again, and we know that all edges have been extracted.

In the most general variation, edge nodes of the winged-edge data structure also include the identifiers *fcw* and *fccw* of its neighbor faces, and analogously to *ncw* and *nccw*, the identifiers *pcw*, *pccw* of the previous edges in these faces. The orientation indicators of Figure 6.6 now become redundant. The resulting "full" winged-edge data structure is shown in Figure 6.7. The schematic diagram (a) indicates once more the meaning of the various data items.

Because the full winged-edge data structure includes the identifier of an adjacent edge into each vertex node, all edges meeting at a vertex can be extracted by an algorithm similar to the loop extraction algorithm (see Figure 6.7(a)).

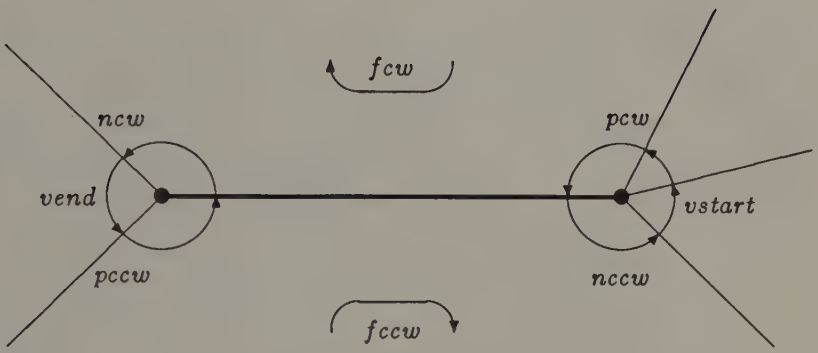
In fact, the data structure treats faces and vertices in a completely symmetric fashion. If the nodes *vstart* and *vend*, and the nodes *fcw* and *fccw* are exchanged, we end up with the *dual* of the original polyhedron.

### 6.2.4 Faces With Several Boundaries

The data structures described so far assume that each face is *simply-connected*, i.e., that it has just one boundary curve. However, sometimes it seems natural to include faces that have several boundary curves; see, for instance, the object depicted in Figure 6.7.

Two alternatives present themselves. One approach is to model faces with nonsimple boundaries by connecting the boundary segments with auxiliary edges. (In [134], this representation was termed the "bridge edge representation".) In the case of Figure 6.8, this could take place by adding the dashed edge. Unfortunately, these special edges should be marked somehow to avoid their inclusion in line drawings. However, in the winged-edge data structure this is not necessary because auxiliary edges are considered to occur twice in the same face, and can be detected by comparing the *fcw* and *fccw* data items.

The other approach is to add a separate loop node that models a simple boundary into the boundary model, and associate each face with a list of loops. References to faces would be replaced by references to the appropriate loops; for instance, face identifiers *fcw* and *fccw* in edge nodes of the full winged-edge data structure would be replaced with the analogous loop identifiers *lcw* and *lccw*. All data structures discussed can be readily generalized in this fashion, and we will not elaborate on this topic further.




(a)

| edge | vstart | vend | fcw | fccw | ncw | pcw | nccw | pccw |
|---|---|---|---|---|---|---|---|---|
| $e_1$ | $v_1$ | $v_2$ | $f_1$ | $f_2$ | $e_2$ | $e_4$ | $e_5$ | $e_6$ |
| $e_2$ | $v_2$ | $v_3$ | $f_1$ | $f_3$ | $e_3$ | $e_1$ | $e_6$ | $e_7$ |
| $e_3$ | $v_3$ | $v_4$ | $f_1$ | $f_4$ | $e_4$ | $e_2$ | $e_7$ | $e_8$ |
| $e_4$ | $v_4$ | $v_1$ | $f_1$ | $f_5$ | $e_1$ | $e_3$ | $e_8$ | $e_5$ |
| $e_5$ | $v_1$ | $v_5$ | $f_2$ | $f_5$ | $e_9$ | $e_1$ | $e_4$ | $e_{12}$ |
| $e_6$ | $v_2$ | $v_6$ | $f_3$ | $f_2$ | $e_{10}$ | $e_2$ | $e_1$ | $e_9$ |
| $e_7$ | $v_3$ | $v_7$ | $f_4$ | $f_3$ | $e_{11}$ | $e_3$ | $e_2$ | $e_{10}$ |
| $e_8$ | $v_4$ | $v_8$ | $f_5$ | $f_4$ | $e_{12}$ | $e_4$ | $e_3$ | $e_{11}$ |
| $e_9$ | $v_5$ | $v_6$ | $f_2$ | $f_6$ | $e_6$ | $e_5$ | $e_{12}$ | $e_{10}$ |
| $e_{10}$ | $v_6$ | $v_7$ | $f_3$ | $f_6$ | $e_7$ | $e_6$ | $e_9$ | $e_{11}$ |
| $e_{11}$ | $v_7$ | $v_8$ | $f_4$ | $f_6$ | $e_8$ | $e_7$ | $e_{10}$ | $e_{12}$ |
| $e_{12}$ | $v_8$ | $v_5$ | $f_5$ | $f_6$ | $e_5$ | $e_8$ | $e_{11}$ | $e_9$ |

| vertex | first edge | coordinates |
|---|---|---|
| $v_1$ | $e_1$ | $x_1\ y_1\ z_1$ |
| $v_2$ | $e_2$ | $x_2\ y_2\ z_2$ |
| $v_3$ | $e_3$ | $x_3\ y_3\ z_3$ |
| $v_4$ | $e_4$ | $x_4\ y_4\ z_4$ |
| $v_5$ | $e_9$ | $x_5\ y_5\ z_5$ |
| $v_6$ | $e_{10}$ | $x_6\ y_6\ z_6$ |
| $v_7$ | $e_{11}$ | $x_7\ y_7\ z_7$ |
| $v_8$ | $e_{12}$ | $x_8\ y_8\ z_8$ |

| face | first edge |
|---|---|
| $f_1$ | $e_1$ |
| $f_2$ | $e_9$ |
| $f_3$ | $e_6$ |
| $f_4$ | $e_7$ |
| $f_5$ | $e_{12}$ |
| $f_6$ | $e_9$ |

(b)

**Figure 6.7** The full winged-edge data structure.

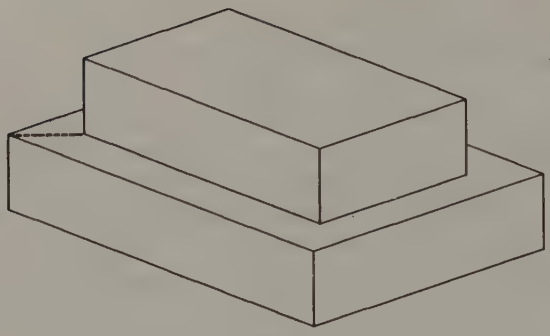


**Figure 6.8** Object with non-simple faces.

## 6.3 VALIDITY OF BOUNDARY MODELS

A boundary model is *valid* if it defines the boundary of a "reasonable" solid object. According to the theory of Section 3.5, we are usually only interested on objects bounded by closed, orientable surfaces. In that case, the validity criteria of a boundary model includes the following conditions:

1. The set of faces of the boundary model "closes," i.e., forms the complete "skin" of the solid with no missing parts.

2. Faces of the model do not intersect each other except at common vertices or edges.

3. The boundaries of faces are simple polygons that do not intersect themselves.

The first and second conditions exclude self-intersecting objects such as the object of Figure 6.9(a). Observe that the pyramid-shaped bottom surface of the object intersects the top surface. The third condition disallows objects such as the "open box" (b). (To represent "open" objects with a solid boundary model, the best approach is to model "openings" as specially marked faces, instead of leaving them out.)

The first condition relates to the *topological integrity* of a boundary model. The theory of plane models of Section 3.5 tells us that all topological integrity criteria can be enforced just by structural means. In particular, according to Definition 3.8, the first condition can be enforced by demanding that each edge occurs in exactly two faces; hence, no edge can

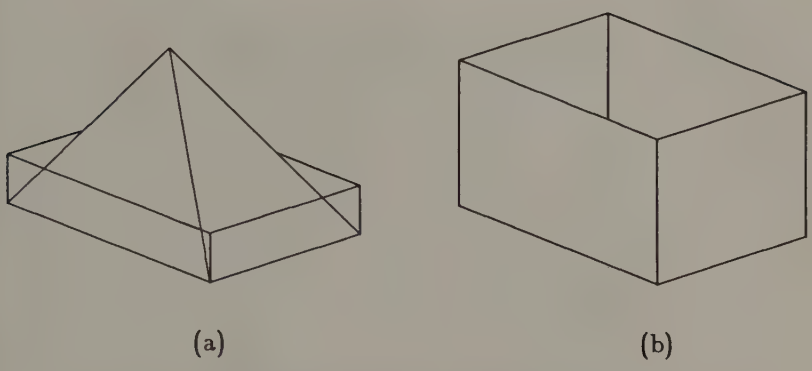



**Figure 6.9** Invalid boundary models.

be the boundary of a missing part of the surface. Observe that the winged-edge data structure "automatically" satisfies this criterion, because edges occurring in just one face cannot be represented.

The winged-edge data structure also enforces that the Möbius' rule of Definition 3.9 is satisfied; in fact, the faces will always be consistently oriented. Hence the surface represented by it is guaranteed to be orientable. However, nonorientable objects are disallowed also by the second condition above, because they would always intersect themselves in the three-dimensional space.

Unfortunately, the *geometric integrity* of a boundary model defined by the second and third conditions cannot be enforced just by structural means: by assigning inappropriate geometric information to a completely reasonable topological entities, invalid models can be created. To guarantee the geometric integrity, one must either resort to a computationally expensive test that involves a comparison of each pair of faces in the solid, or limit the user's freedom by giving him only validity-enforcing solid description mechanisms.

## 6.4 DESCRIPTION OF BOUNDARY MODELS

Examination of Figures 6.6 and 6.7 soon reveals one of the major problems of boundary models, namely the complexity of their construction. It is beyond the capabilities of a normal human being to build correct boundary models directly, e.g., by typing information such as that of Figure 6.7.

The designer of a boundary modeler is therefore faced with the need of providing a sufficient collection of more convenient and efficient solid de-

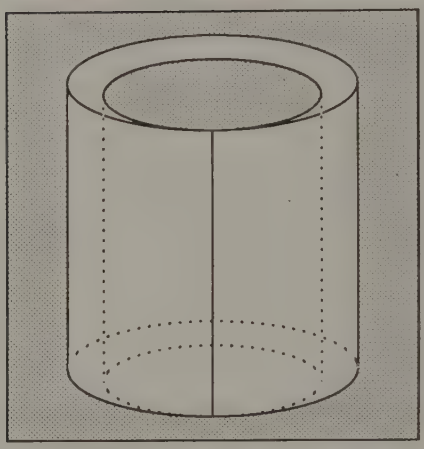


**Figure 6.10** An object with no convenient BR.

scription facilities. On the other hand, he must take the integrity problems of boundary models discussed in the previous section into account, and seek for solid description techniques that either guarantee the integrity criteria, or (al least) make it difficult to construct invalid models.

Some frequently used description mechanisms are discussed briefly below.

### 6.4.1 Conversion from CSG

A common solution is to give a CSG-based solid description language to the user and construct boundary models through conversion from CSG. In this approach, the boundary modeler must include algorithms for creating boundary models from the CSG primitives, and an algorithm for Boolean set operations on boundary models.

Unfortunately, set operations for boundary models are computationally expensive and sensitive for numerical problems. Even if the numerical problems of calculating surface intersections can be solved, the problem of the mismatch between the modeling spaces of CSG and conventional boundary models remains. That is, some objects that can be represented in CSG have no convenient representations as boundary models.

An example of this is shown in Figure 6.10. The object can be represented in CSG as the difference of two cylinders. On the other hand, ordinary boundary data structures (such as the winged-edge data structure)

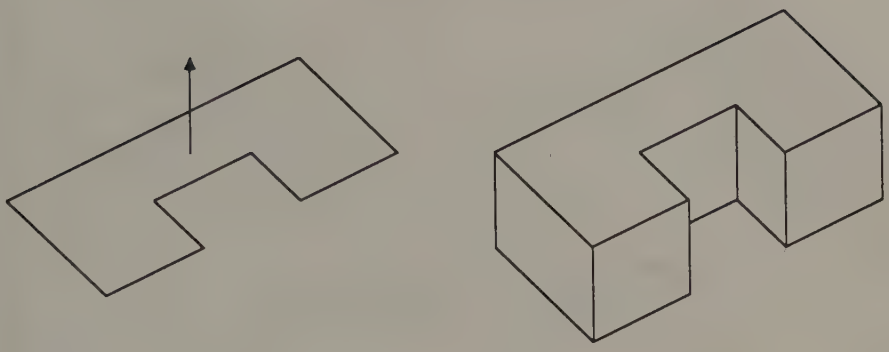


**Figure 6.11** Solid description by drawing and sweeping.

cannot represent properly the region where the "inner" cylinder touches the "outer" one. Because an edge cannot have four neighbor faces, a pair of coinciding edges must be used. This, however, breaks down the geometric integrity criteria, and problems may arise if additional Boolean operations are performed on the boudary model.

Reference [96] outlines a number of set operations algorithms for boundary models; we shall study the problems of set operations further in Chapter 15.

### 6.4.2 Two-and-Half-Dimensional Drawing

Many objects needed in practice exhibit symmetry that can be exploited in their description. Quite often the object can be described in terms of a two-dimensional cross section, and information of material "thickness." Rotationally symmetric objects give another example of how the solid can be described just by a profile and a rotation axis.

Sometimes both of these types of objects could be described in terms of basic objects and Boolean set operations. Nevertheless, many people find it far more natural to "draw" an outline, and "sweep" it to give it a thickness or "swing" it to make it a rotational solid. Because of the relatively large computations needed for evaluating Boolean operations of boundary models, these operations can be executed much more efficiently than the sequence of set operations that creates the same model.

These *two-and-half-dimensional* solid description mechanisms can naturally be incorporated to boundary modelers. The term "two-and-half-dimensional" emphasizes that most description operations take place in

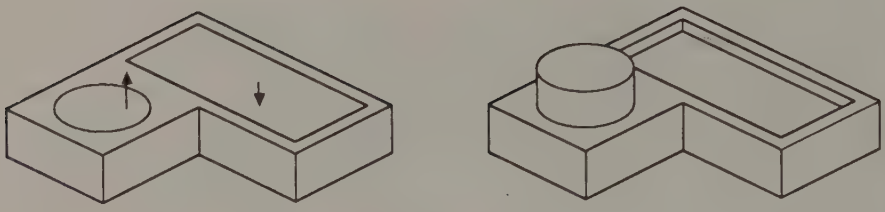


**Figure 6.12** Face sweeping.

two dimensions, and can be visualized as operations on a two-dimensional graphical model.

One possibility is to include separate sweeping primitives in the boundary modeler, and to provide a conversion algorithm from the sweeping model to a boundary model. This approach is completely analogous to the inclusion of sweeping primitives to CSG models described in Section 5.2.

A more general approach is to consider sweeping an operation that can be performed on *any* planar face of a boundary model. This "face lifting" operation allows, for instance, the construction sequence of the object depicted in Figure 6.12. This efficient and economical input style is heavily used in boundary modelers belonging to the so-called BUILD group of solid modelers [53].

Engineering drawings generally display an object from several orthogonal views. This technique can be used for describing boundary models in the form of the so-called *profile set operations*. This operation combines two (or several) closed profiles, representing the outlines of an object from two views, by sweeping the outlines by an appropriate amount and evaluating the Boolean intersection of the swept objects. Surprisingly many objects can be modeled quite satisfactorily with this operation only; see, for instance, Figure 6.13.

Unfortunately, the sweeping operations are not completely "safe" either; is is easy to create self-intersecting objects with them, for example. See [18] for discussion on conditions under which sweeping and similar operations can be kept integrity-preserving.

### 6.4.3 Local Modification

A third natural solid description facility is based on *local modifications* of a boundary model. Actually, even the face lifting operation of the preceding section can be understood as a local modification that replaces a face by a

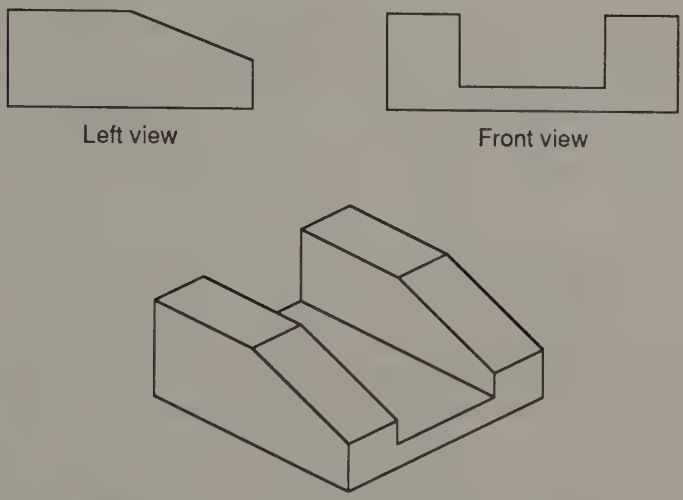



**Figure 6.13** A sample profile set operation.

collection of new faces modeling the lifted region.

The boundary modelers BUILD-2 [18] and ROMULUS include a very rich collection of local modification operations. A typical example is the "rounding" of a sharp edge as depicted in Figure 6.14(a). A more general case of this is the "blending" of several adjacent edges depicted in (b); note how a vertex is replaced with a triangular blend surface.

The boundary modeler MODIF [25] includes another kind of a local modification operation. In MODIF, a solid is first modeled as a polyhedron. The polyhedron is then converted into a curved surface model by introducing control points on each edge. The user is then allowed to modify the coordinates of control points in order to create the desired shape. Special techniques are employed to keep the surface smooth.

## 6.5 ALGORITHMS FOR BOUNDARY MODELS

### 6.5.1 Visualization

The design of visualization algorithms for boundary models is greatly simplified by the availability of the faces, edges, and vertices of a solid. In this respect, they may be considered "explicit" models in comparison to CSG models that must be "evaluated" for visualization purposes.

All techniques for generating graphical output of graphical models are directly applicable to boundary models. In addition, for a polyhedral model

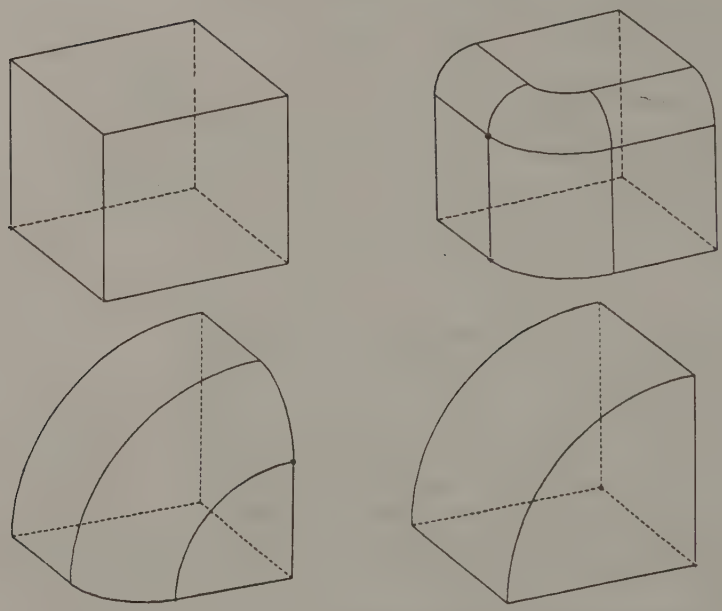


**Figure 6.14** Local modifications of boundary models.

all well-known techniques of hidden line and surface removal and shading can readily be applied, including "exact" object-space methods. Furthermore, advanced graphical display devices often include hardware support for the rapid processing of polyhedral models.

Unfortunately, the presence of curved surfaces makes things far more complicated especially if hidden line output is required. There are specialized hidden line removal algorithms that can operate also on natural quadratic surfaces (see [70]), but in general it is more efficient to convert curved-surface models into polyhedral ones in order to create hidden line output.

Hidden surface removal is (surprisingly) simpler, because standard techniques such as *scan-line algorithms*, the *z-buffer algorithm*, and ray casting are applicable. These methods allow the generation of very high quality images at medium to high computational cost.

### 6.5.2 Integral Properties

Two general techniques are available for the calculation of basic engineering properties based on boundary models, namely the method of *direct integration* and the use of the *divergence theorem* of calculus [69].

Direct integration is the standard technique discussed in calculus textbooks. This technique is based on evaluating a volume integral over a solid as the sum of the appropriately signed contributions of its faces; see Figure 6.15(a).

For instance, the volume integral of a function $f(x, y, z)$ over a solid $S$ can be evaluated by

$$\int\int\int_S f(x,y,z)dzdydx = \pm\sum_i \int\int_{F_i'} \left( \int_0^{z_i(x,y)} f(x,y,z)dz \right) dydx \tag{6.1}$$

where $F_i'$ is the projection of face $F_i$ on $xy$-plane, and $z_i(x, y)$ is obtained by solving the equation of $F_i$ for $z$. The sign is determined by looking at the surface normal of the face: if the normal is facing the $xy$-plane, the minus is selected, otherwise the plus.

The resulting double integral can be evaluated by applying a similar technique to evaluating the contribution of each edge of $F_i$; see Figure 6.15(b). That is, the area integral of a function $g(x, y)$ over the face can be evaluated by

$$\int\int_{F_i'} g(x,y)dydx = \pm\sum_j \int_{E_j'} \left( \int_0^{y_j(x)} g(x,y)dy \right) dx \tag{6.2}$$

where $E_j'$ is the projection of $E_j$ on $x$-axis, and $y_j(x)$ the equation of $E_j$ solved for $y$. This will finally lead us to single integrals of the form

$$\int_{x_{j,1}}^{x_{j,0}} h(x)dx. \tag{6.3}$$

where $x_{j,0}$ and $x_{j,1}$ are the (projected) endpoints of $E_j$.

Observe that both $z_i(x, y)$ and $y_j(x)$ are assumed to be single-valued in $x$ and $y$. Hence curved faces and edges may have to be subdivided. It may still be hard to solve the surface and curve equations for $z$ and $y$ even numerically, and the equations of the projected edges $E_j'$ of $F_i'$ may not even be available. However, for polyhedral models the direct integration is quite attractive, because face equations are either single-valued or can be ignored, and straight lines project to straight lines.

The divergence theorem provides an alternative method for the evaluation of integral properties [122]. From the observation that it is always possible to find at least one vector function $g(x, y, z)$ such that $\mathrm{div} g = f$ for any given continuous function $f(x, y, z)$, it follows that

$$\int_S f\, dV = \int_S \mathrm{div} g\, dV = \sum_i \int_{F_i} g \cdot n_i\, dF_i$$

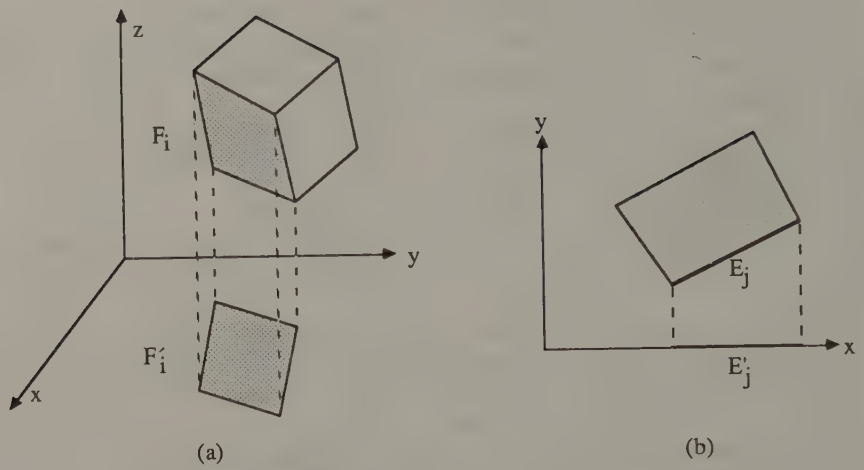



**Figure 6.15** The method of direct integration.

where $F_i$ is a face of the solid, $S$, $n_i$ is the unit normal vector to $F_i$, and $dF_i$ is the surface differential. For instance, if

$$f(x, y, z) = y^2 + z^2,$$

then, for example, the choices

$$g(x, y, z) = [x(y^2 + z^2) \;\; 0 \;\; 0]$$

and

$$g(x, y, z) = [0 \;\; y^3/3 \;\; z^3/3]$$

are both possible. If all $F_i$ are planar polygons, integrals on the right side can be readily evaluated. In the general case, approximation techniques are needed; see [122] for additional information and practical results.

In addition to the general methods discussed above, very straightforward direct techniques for evaluating integral properties of consistently oriented polyhedral objects are available. For instance, the volume of a polyhedron can be reduced to calculating to the sum of the signed volumes of tetrahedra [129].

Assume the tetrahedron has one corner at origin, and the other corners at

$$P_i = [x_i \; y_i \; z_i],$$

for $i = 1, 2, 3$. Then the signed volume of the tetrahedron is given by

$$V_{tetra} = \frac{1}{6} \; P_1 \times P_2 \cdot P_3$$

where $\times$ and $\cdot$ stand for the vector and scalar product of vectors. Observe that $V_{tetra}$ is positive or negative depending on whether the orientation of the triangle $P_1 \; P_2 \; P_3$ as viewed from the origin is clockwise or counterclockwise. The volume of a polyhedron can calculated by taking all tetrahedra formed by three consecutive vertices in faces of the solid, and summing their signed volumes. The reference [129] reports very similar algorithms for other integral properties.

## 6.6 PROPERTIES OF BOUNDARY MODELS

We can again formulate the summary of this chapter by discussing the properties of boundary models within the general framework of Section 3.6.1.

*Expressive power*: The modeling space of boundary models depends on the selection of surfaces that can be used. There is no inherent reason to limit this collection to mere half-spaces; hence boundary models can be used to represent objects from a more general modeling space than that possible for CSG.

On the other hand, as noted in Section 6.4.1, some objects that can be represented in CSG have no convenient representations as boundary models. While knowledgeable users can avoid such objects when working in direct interaction with a boundary modeler, this deficiency may cause problems e.g. if conversion from a CSG model is attempted.

*Validity*: Validity of boundary models is in general quite difficult to establish. Validity criteria split into topological constraints and geometric constraints as discussed above. While it is possible to manage topological validity without large overhead, it is hard to enforce geometric correctness without penalizing the speed in interactive design.

*Unambiguity and uniqueness*: Valid boundary models are unambiguous. They are not unique.

*Description languages*: Boundary models are tedious to describe directly. Fortunately, it is possible to build description languages that are based on graphical "drawing" and "sweeping" operations or on a CSG-like input on top of a boundary model.

*Conciseness*: Boundary models of useful objects may become large, especially if curved objects are approximated with polyhedral models.

*Closure of operations*: As discussed in Chapter 3, boundary models are usually *not* closed under set operations (regularized or not). In this theoretical sense boundary model description mechanisms based on CSG

conversion or set operations are always vulnerable. The natural closed operations for boundary models are the *Euler operators* to be studied in Chapter 9.

*Computational ease and applicability*: Boundary models are useful for generating graphical output, because they readily include the data needed for driving a graphical display. Analysis algorithms based directly on boundary models become quite difficult if representations of objects from more general modeling spaces than polyhedra must be processed.

We shall return to boundary models in much more detail in Part Two of this book.

## PROBLEMS

6.1. Describe an algorithm for extracting the edges around a vertex from the full winged-edge data structure of Figure 6.6.

6.2. Form the dual of the full winged-edge data structure of Figure 6.6 as indicated in the last paragraph of Section 6.2.3. Which object do you get?

6.3. Present (in the style of Figure 6.6) a modified winged-edge data structure of the "pyramid" of Figure 6.7 that can represent faces with several boundaries.

6.4. Based on the general Equations 6.1–3, outline an algorithm for evaluating the following integral properties of a polyhedral solid $S$:

(a) the volume $V$, which is expressed as the triple integral

$$V = \int\int\int_S dzdydx$$

(b) the moment of inertia about $z$,

$$I_z = \int\int\int_S x^2 + y^2 \, dzdydx$$

## BIBLIOGRAPHIC NOTES

An overview of boundary data structures and the representations used in a number of modelers is given in the survey by Baer, Eastman, and Henrion [8], which still is recommended reading for a student of solid modeling.

The solid modelers BUILD-1 and BUILD-2 written by the former BUILD-group at the University of Cambridge under the leadership of Ian Braid belong to the most influential works to the solid modeling technology. The report [18] provides an overview of the design and implementation of the version of BUILD-2 of that

time, and is a must to anybody contemplating the design of a boundary modeler. Work on BUILD modelers still continues at the Cranfield Institute for Technology.

A good reference to scan-line techniques is [102].

Section 6.5.2 is based on the survey article on integral properties by Lee and Requicha [69].

# Chapter 7

# HYBRID MODELERS

None of the three major approaches to solid modeling we have described so far is superior to the others in all respects. This motivates the use of multiple simultaneous representations in a serious modeling system. These *hybrid modelers* are the topic of this chapter.

## 7.1 WHY MULTIPLE REPRESENTATIONS?

In the three previous chapters, the three major approaches to solid modeling were introduced, and their inherent properties discussed. Even at the cost of oversimplification, let us review some key points of this discussion:

1. The simplest of the three approaches, *decomposition models* are superior to the others as sources of data for numerical algorithms, even when they are approximative. Except when the original source of information is a binary image, their creation is complicated without conversion from other representations. They are also voluminous.

2. *Constructive models* are the most concise of the three major approaches to solid modeling. Because the directly available methods for the generation of graphical output or the creation of data for numerical algorithms are slow, these tasks are performed through conversion to other representations.

3. *Boundary models* are directly useful for graphical applications. Especially polyhedral models are useful for numerical applications as well, if the limited modeling space captured by polyhedra can be tolerated. Without the help of other representations and conversions, their creation is difficult, however.

Clearly, a *combination* of the three major approaches to solid modeling would be superior to any of them alone. This has led designers of solid modeling systems to investigate possibilities of combining the basic approaches.

A *hybrid* solid modeler is capable of supporting several coexisting solid representations, and tries to pick the most suitable of them for each task. These representations are guaranteed to be *consistent* up to the level made possible by differences in the inherent modeling spaces. Consistency enforcement is materialized in *conversion algorithms* that can go from one representation to another.

Hence, the representations stored by a hybrid modeler are a combination of several different basic representations. A graphical model can well appear as one of these representations. Observe that a hybrid modeler can well include, say, several types of boundary models. For instance, it may store a winged-edge representation with curved surfaces for modeling procedures, and a distinct polyhedral model for display purposes. Solid representations for secondary storage could also form still another distinct boundary model.

## 7.2 PROBLEMS OF HYBRID MODELERS

The coexistence of several solid representations in one modeler raises a number of new kinds of problems.

### 7.2.1 Conversions

Clearly, the conversion algorithms form a fundamental part of a hybrid modeler. Unfortunately, there are inherent limitations on what kinds of conversions a hybrid modeler can expect to support.

Constructive models and CSG models in particular have the virtue that they can be converted to any other representation. The feasibility of the conversion to a boundary model, the *boundary evaluation* has been well demonstrated in many research and commercial CSG modelers. General approaches for converting from a CSG representation to an enumerative representation were outlined in Chapter 5.

Unfortunately, the inverse conversions are not available. In particular, the *inverse boundary evaluation*, i.e., the conversion from a boundary representation to a CSG model, still remains an unsolved problem for practical purposes in the general 3-dimensional case. However, boundary models can be converted to decomposition models approximately as easily as CSG.

The conversions from a decomposition model to a boundary model or a CSG model would be valuable in the case that information received as

a binary image must be mapped against information stored as a boundary model or a CSG model. Robot vision offers an example of great practical interest. Unfortunately, these conversions are still more a research subject than a technology that could be exploited in practical solid modelers.

Because conversions from boundary modes or CSG models to decomposition models generally are approximative, it may seem that the inverse conversion makes little sense: if the exact information is already available, why attempt its reconstruction? Solid modelers based on boundary models or CSG models usually assume this view, and treat decomposition models as *transient* representations, i.e., decomposition models are created "on the fly" to solve certain problems, and discarded afterwards.

In fact, *all* modelers that create graphical output during an interactive design session can be considered hybrid modelers, because the display is a kind of geometric model of its own.

### 7.2.2 Consistency

If a conversion algorithm between two representations included in a hybrid modeler is available, it will be possible to keep the representations consistent. Unfortunately, this usually happens at the cost of limiting the functionality available in the modeler.

For instance, CSG models cannot ordinarily include parametric surfaces. This means that if a boundary model created by conversion from CSG is modified further to include blends represented as parametric surfaces, the original CSG representation of the object modeled cannot be kept consistent with it.

In general, a hybrid modeler that aims at total consistency between the representations can only support such operations on solids that can be mapped to each of the representations. Boolean set operations have their prominent position in solid modeling partly because they can be mapped to all the major solid representations. Unfortunately, set operations alone are not sufficient for constructing a good user interface to a solid modeler, and it is much more difficult to support "drawing" operations on several representations.

### 7.2.3 Modeling Transactions

All modifications to solid representations take place in a sequential fashion. Therefore, a total consistency between the various representations cannot exist at all time instances.

This leads us to define the concept of a *modeling transaction* as an indivisible sequence of modeling operations that commences from a consis-

tent state and ends in a consistent state. For instance, in a hybrid modeler consisting of a CSG modeler and a polyhedral modeler, the modeling transaction of performing a set operation on two primitives would consist of the creation of a CSG tree consisting of the primitives, and computing a polyhedral model of the result of the set operation. In an interactive environment, the display can be considered a representation in its own right; in this case, the modeling transaction should include the updating of the displayed image. Solid representations in secondary storage add another dimension to modeling transactions: an interactive modeling session is complete only after the relevant results are stored.

The functions available to the user must be designed with care in any solid modeler. In a hybrid modeler the careful design becomes still more important.

## 7.3 HYBRID ARCHITECTURES

The solution usually adopted in hybrid modelers to the problems of lacking conversions and consistency is to treat one of the representations supported as the *primary* representation, and the others as *secondary* representations.

In this respect, most solid modelers available today seem to fall into one of two major architectures. The modelers in the first category (Figure 7.1(a)) ordinarily use CSG trees as their primary solid representations. From CSG trees, boundary models may be created through boundary evaluation, and decomposition models through classification algorithms in the style discussed in Section 5.2.

The user, however, has no direct access to the secondary representations. In other words, although boundary models might be included in the modeler (and perhaps continuously updated through incremental boundary evaluation to reflect changes in the CSG tree), the user of the modeler cannot perform modeling operations which are specific to a boundary model, say, local modifications. Modelers such as PADL-1 [127], PADL-2 [21], and GMSOLID [17] belong to this group.

Modelers that use boundary models as their primary representation form the second group (Figure 7.1(b)). These modelers usually include CSG as one solid description facility, and many of them also store a variation of the CSG tree (see e.g. [18]). In addition to CSG, however, these modelers have also other solid description facilities that directly modify boundary data structures. These facilities may include the "drawing" and "sweeping" operations or local solid modifications as discussed in Chapter 6. Hence, the role of CSG in these systems is really that of an auxiliary representation only. Many industrially used modelers, such as ROMULUS,

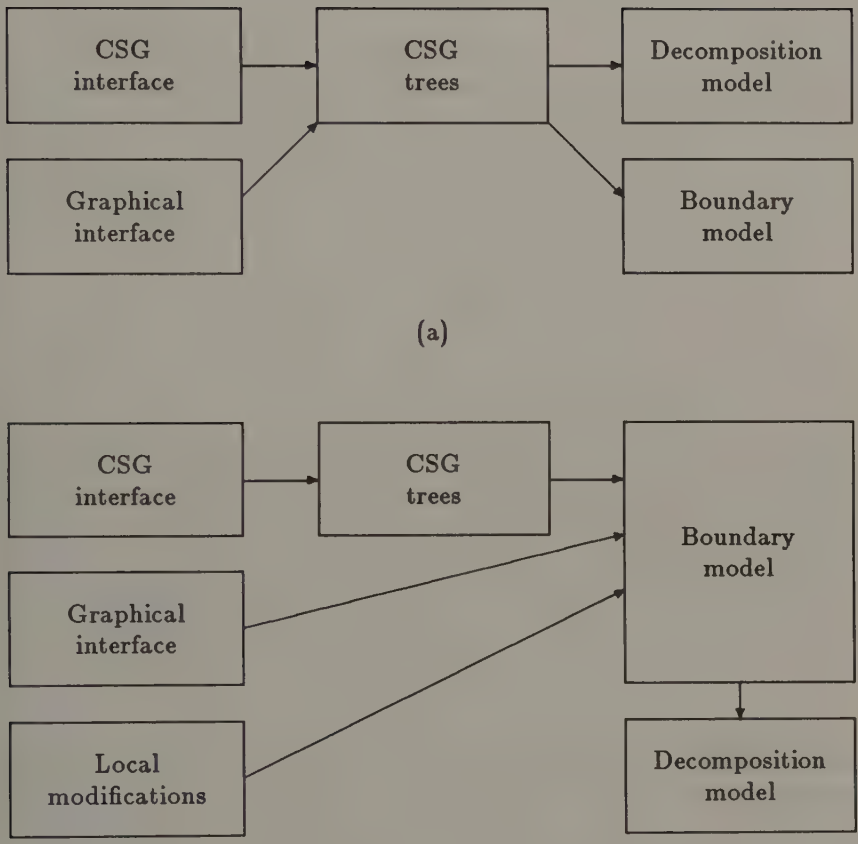



**Figure 7.1** Architectures for hybrid modelers.

GEOMOD [118], and MEDUSA [87] belong to the second major group.

Various architectures for hybrid modelers are discussed in greater detail in references [99,100], including information of numerous solid modelers.

## 7.4 DISTRIBUTED MODELERS

The current trend towards the use of *distributed* graphics systems consisting of a host computer and graphics workstations has significance to hybrid solid modelers. As current graphics workstations have considerable processing power of their own, and actually are usually small computers themselves, the problem of dividing the labor sensibly between the host and the workstations becomes urgent. The analogous problem also arises in top-of-the-line workstations that have a special graphics processor for the rapid display of polygonal models.

A solution to this dilemma is the introduction of a *distributed modeler*, part of which resides in a workstation and part in the host. Because the modeler maintains solid models in both parts, distributed modelers are a special case of hybrid modelers.

As an example of the division of labor in a distributed modeler in the context of CSG models, Atherton [6] suggests the separation of a *visual modeler* from a *analytical modeler*. The visual model is intended to support rapid interactive processing, and resides locally in the workstation. The analytical model is stored in the (still quintessential) host computer; for CSG, the analytical model is the exact CSG tree. In particular, Atherton suggests that CSG primitives are approximated by polyhedra in the workstation, and proposes an efficient algorithm for rapidly displaying CSG trees consisting of such polyhedral primitives.

A similar division of labor is also possible in boundary models. So-called *faceting modelers* are a class of boundary modelers that "know" about curved surfaces, but do not represent them explicitly. Rather, they store polyhedral approximation of the surface, and it is this representation that is used for, say, generation of graphical output. The distinction between faceting modelers and what we have called merely polyhedral modelers is that polyhedral modelers do not know where the planar faces came from, even if they can create polyhedral approximations of curved objects.

Many graphics workstations store a structured display list with polygonal graphical primitives, which can be considered a simple polyhedral model. The display list is constructed by a boundary modeler residing in the host computer, and it must be updated to reflect operations performed on the solid model. Unfortunately, this architecture still poses heavy requirements for host-workstation communication in the distributed

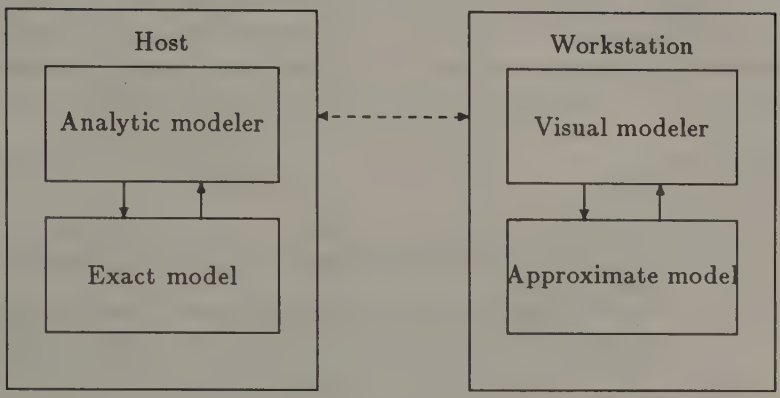



**Figure 7.2** A distributed modeler.

environment.

As a step forward, a faceting modeler could be used in the role of a visual modeler, i.e., to provide rapid interactive processing locally in a workstation. This approach is strongly supported by that many graphics workstations have special hardware for rapid processing of polyhedral models. For numerical applications requiring high accuracy, the implicit surface information of the faceted modeler can be transmitted to the host computer, and elaborated by the analytic modeler.

## 7.5 OTHER HYBRID APPROACHES

Hybrid modelers discussed in the previous sections use several coexisting independent representations to be able to use the most appropriate ones for each task.

To avoid the inherent difficulties caused by several represenrations, modelers that pursue the combination of several basic approaches in a single integrated solid representation scheme have been proposed.

Typically, these modelers combine a decomposition model (exhaustive enumeration or an octree) with either a CSG model or a boundary model, and use the decomposition model as a geometric access method to CSG or boundary information.

For instance, the *extended octree* of Ayala *et al.* [7,86] is based on storing a boundary representation of a polyhedral object within the cells of an octree. The octant subdivision is continued until each cell contains at

most one vertex, one edge, one face, or is homogeneously "full" or "empty." The cited references give algorithms for conversion from a boundary representation to an extended octree and vice versa, and for set operations of extended octrees. References [22,136] describe similar representations.

## PROBLEMS

7.1. Are there other solid description techniques besides set operations that can easily be supported on top of all major approaches to solid modeling?

7.2. The separation between "general-purpose" geometric models and graphical data structures (e.g., structured display lists) for display purposes is debatable. Consider the pros and cons of keeping these data separate versus integrating them into a single representation.

## BIBLIOGRAPHIC NOTES

An algorithm for inverse boundary evaluation is expected to convert an arbitrary boundary model into an equivalent CSG tree. Unfortunately, the problem seems to be subject to combinatorial explosion, and rather difficult to solve optimally (i.e., with the minimum number of primitives) or even satisfactorily.

The feature extraction algorithm of Woo [130] can *in principle* generate a CSG expression from a boundary model by using a convex hull technique. Unfortunately, without human help the algorithm seems to be subject to infinite looping.

Two-dimensional versions of the problem are much simpler. See, for instance, the work of Vossler [128] on the practically interesting problem of the conversion of swept planar outlines consisting of straight lines and circular arcs to a CSG model.

As for the other conversions, Kunii *et al.* [66] have reported an algorithm for converting an octree to a boundary representation. However, the algorithm makes no attempt to infer the actual surfaces of a curved object from the rasterized data.

# Part Two

# THE GEOMETRIC WORKBENCH

Part One covers the basic approaches and technologies of solid modeling without going deeply into system and implementation aspects. In contrast, Part Two takes a much closer look boundary modeling by means of developing the basic algorithms for the *Geometric WorkBench* (GWB), a simple polyhedral solid modeler following the boundary approach.

# Chapter 8

# TOWARDS THE GEOMETRIC WORKBENCH

Before we go deeply into the development ot the Geometric WorkBench, it is useful to take a bird's-eye-view look into its basic ideas and its architecture, and give an outline of how the following chapters relate to its overall structure.

## 8.1 DESIGN GOALS OF GWB

Solid modeling systems are certainly fairly nontrivial programs, and ideally their construction should be based on very well understood methods and a very clear overall conception of the software architecture of the modeler. Because we shall attempt to describe the construction of a solid modeler in the limited space available in a few chapters of a book, these requirements are even more important for GWB.

What characterizes a "good" software design, then, and what design and development methods should be followed to guarantee the quality of the result? These problems form the nutshell of the field of *software engineering*, and a reader not familiar with it can certainly benefit from getting acquaintained with software engineering literature (e.g., [113] or the "classic" [20]).

While many software design and development methodologies have been described, and some even applied succesfully in practical work, a focal point of the various alternative approaches can be labeled with the term *modu-*

*larity.* A complicated problem can only be solved if it can be broken into simpler subproblems that can be solved separately, and later combined to yield the total solution. To allow for division of labor in software design and development, the subproblems should be relatively independent and have simple, "narrow" interfaces to each other. If the components of a software system have these properties, they are said to exhibit "good modularity."

Another key expression of software engineering methodology is *abstraction.* The inner workings of a significant software system can be exceedingly complicated, and a human being cannot perceive the logic of its operation if he/she attempts to understand all its details at once, much less design or implement it. When breaking a problem down into modular subproblems, it is therefore vital to define the intended operation of each component in abstract terms, and consider the internal details of its implementation separately from the abstract external specification.

Quite properly, the development of various *abstraction mechanisms* for the specification and implementation of software modules has been one of the major subareas of software engineering. It is beyond the scope of this book to review these mechanisms in detail here. Suffice it to say that a major goal of the various abstraction mechanisms is the *information hiding*: a good module should appear opaque to the external viewer as far its internal operation is concerned, and certainly not exhibit surprising side effects.

When applied to a collection of modules, one particular approach to achieve information hiding is the division of the software system into *layers*, each layer consisting of modules with a clear abstract meaning such that the layer forms a functionally complete "abstract machine." If this approach is strictly followed, layer $i$ will use the facilities made available by the layer $i-1$, but has no access to any of the layers $i-2$, $i-3$, and so on. In turn, layer $i+1$ will be based on the external properties of layer $i$.

## 8.2 ARCHITECTURE OF GWB

In the design of GWB, we shall make an attempt to follow the general principles outlined in the previous section. In particular, we shall try to distuingish between separable layers of modules, each forming a functionally complete "modeling machine" suitable for the implementation of the next layer.

### 8.2.1 Layers of GWB

The overall software architecture of GWB forms the onion-like structure depicted in Figure 8.1. In this architecture, the innermost layer will consist

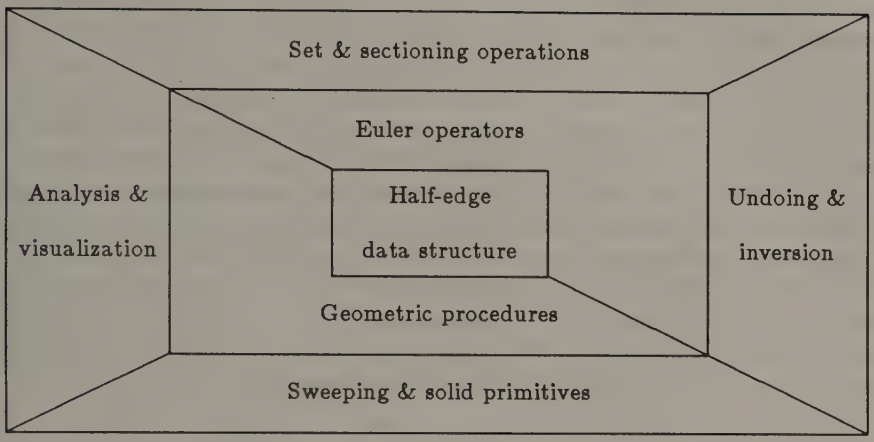



**Figure 8.1** Architecture of GWB.

of the internal data representation of GWB, the *half-edge data structure*, and a set of procedures for its low-level manipulation. The half-edge data structure is a variation of the full winged-edge data structure outlined in Chapter 6. The half-edge data structure is described in detail in Chapter 10.

The next layer will consist of the so-called *Euler operators* to be described in Chapters 9 and 11, and a set of basic geometric procedures to be described in Chapter 12. As to be discussed in more detail later, Euler operators form the basic modeling procedures of GWB that will be used to implement all higher-level operations.

The next layer consists of various "medium-level" modeling tools that make the description of meaningful models more straightforward. These procedures implement a collection of local model manipulations, such as sweeping and "swinging" (rotational sweeping), and allow the construction of various solid primitives. Without a better term, we shall use the term "model description tools" of these procedures. They are the subject of Chapter 13. The model analysis tools introduced in Chapter 12 also belong to this layer.

The following layer is formed by various "model manipulation tools" such as splitting and sectioning (Chapter 14), and Boolean set operations (Chapter 15).

Chapter 16 will consider a useful extension to the Euler operator layer of Chapter 11 that allows the undoing of arbitrary sequences of Euler op-

erators. This extension makes it possible to construct certain useful model manipulation tools, such as procedures for storing models into external storage (the *inversion algorithm*), and retrieving them back.

Using the functionality developed in the earlier chapters, Chapter 17 will describe the final layer of GWB, namely its *user interface* through which the modeling tools of the various levels finally are made available to a user.

Most of the Part Two will deal with the construction of the basic polyhedral modeler with little attention on storing information on curved surfaces, or on providing interfaces towards analytical modelers. The final Chapter 18 will describe extensions to the basic modeler that make it a proper faceting modeler. With these extensions, the functionality of GWB is sufficient for interactive processing in a graphics workstation, and for exchanging exact information with a distinct analytical modeler.

### 8.2.2 An Analogy: Computer Languages

A useful analogy to the architecture of GWB may be found from the relationships of various layers of computer languages.

In this analogy, the half-edge data structure and the low-level operations on it have the role of the processing hardware. Indeed, the facilities available in a particular computer—its registers, memories, and data transfer paths, and the machine language operations that manipulate them—determine the scope of operations that can be performed. Similarly, all modeling operations will finally be mapped onto operations performed on the data representation.

Of course, it would be very inconvenient to describe nontrivial computations directly in terms of a machine language, and a need for higher-level languages arises. On the next level, the assembly language can be used to describe arbitrary computations in a much more convenient notation, which can be mapped into the machine language with the help of an assembler.

In our analogy, Euler operators take the role of the assembly language. Indeed, they describe operations of the underlying "machinery" in a much more convenient and mnemonic notation than a program written to manipulate the half-edge data structure directly.[1] Nevertheless, they are still of a relatively low level, and should be hidden beneath other, more powerful operations.

The analogy works also on higher levels. Instead for assembly language programming, we prefer to work with high-level languages that give us more natural tools for expressing computations than those available on the

[1]Going one step further, Euler operators are a kind of "machine-independent assembler" in that they can be implemented in similar fashion on the top of many different data structures. Well, let's not try to stretch the analogy too far.

assembly language level. Likewise, we shall provide higher level tools for geometric modeling on the top of the Euler operator level. In particular, the user will never use Euler operators directly, but only the higher-level operations built on top of them.

Just as in the case of high-level languages, many alternatives as for these higher levels exist. While we shall concentrate on certain often used high-level modeling tools, it should be borne in mind that these are not the only possibilities, just as the same computer can be used to run both LISP and COBOL, and anything in between and beyond.

## PROBLEMS

8.1. Discuss the benefits of good software engineering practices as for the reuse and maintenance of software.

8.2. The onion-layer structure is just one approach to the construction and management of large software systems. Discuss the potentials of some other approaches you are familiar with (such as object-oriented programming) for a task such as the design and implementation of a solid modeler.

# Chapter 9

# EULER OPERATORS

In the architecture of GWB, the Euler operators have a central role. Based on the theory of plane models of Chapter 3, this chapter takes a closer look on the manipulation of boundary data structures, derives Euler operators, and presents rigorously their properties.

## 9.1 MANIPULATION OF PLANE MODELS

As noted in Section 6.3, the topological integrity of a boundary data structure of the winged-edge type can be enforced just by structural means. Nevertheless, the problem of manipulating a complicated data structure while making sure that all required references to nodes are maintained properly is somewhat difficult, and makes the writing of nontrivial algorithms (such as Boolean set operations) inconvenient.

Fortunately, the theory of plane models will give us help. As discussed in Section 3.5, plane models give us a useful mathematical abstraction of boundary models. In particular, the topological properties of the surface can be evaluated from the plane model. What we still need, however, is a scheme for creating plane models that have some desired properties.

Our aim here is to identify a small set of plane model manipulation operations that are powerful enough to describe *all* plane models of physical significance. While being general, the operations should be *safe* in that, for instance, nonorientable models (such as the Klein bottle of Figure 3.8 on page 42) cannot be created with them.

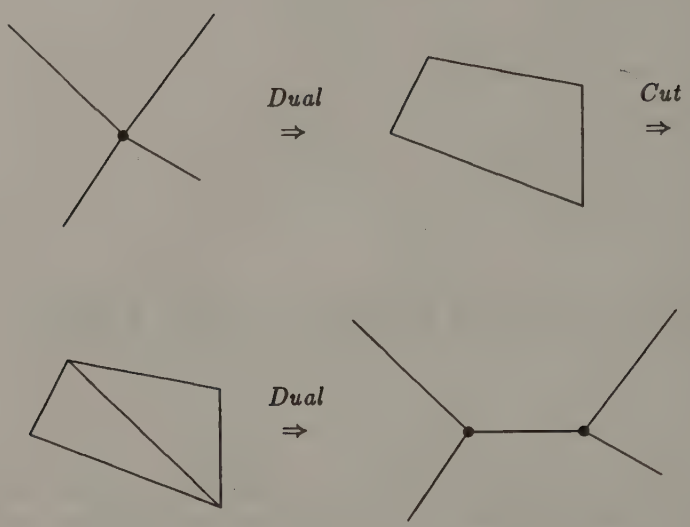



**Figure 9.1** The vertex splitting operation.

### 9.1.1 Local Topological Operations

One of the main results of the plane model theory presented in Section 3.5 is the invariance theorem (Theorems 3.11 and 3.12, pages 43–45). The theorem states that the Betti numbers and all the topological properties conveyed by them (such as the Euler characteristic and orientability) remain invariant for all plane models that can be constructed for a surface.

A direct consequence of the invariance theorem is the following:

**Lemma 9.1** *The topological properties of a plane model are not altered by either cutting a polygon in two or pasting together two polygons along a common edge.*

We interpret the cutting and pasting operations of polygons as a pair of inverse manipulation operations for plane models. That is, the cutting operation can divide a polygon into two polygons by joining two of its vertices with a new edge. Hence the sequence of edges around the polygon is divided into two sequences. Conversely, the inverse pasting operation removes an edge separating two polygons and merges their edge-cycles.

It is natural to ask whether there are other useful manipulation operations. In the search, the duality introduced in Section 3.5.8 offers help. To

see how, let us consider the effect of the cutting operation on the dual plane model. As demonstrated in Figure 9.1, a cutting operation on a polygon of the dual model has the effect of splitting the corresponding vertex of the original model in two vertices. Similarly, a pasting operation on the dual has the effect of a vertex joining operation that combines two vertices connected with an edge into one, and merges their edge-cycles. The splitting and joining operations are inverses of each other.

By Lemmas 9.1 and 3.13 of Section 3.5.8, a polygon cutting operation or its inverse on the dual plane model will preserve all topological characteristics of a plane model. Therefore, the vertex splitting operation or its inverse do not alter the topological properties of the plane model they modify, and they can be added to our set of manipulation operations.

Do we need other operations? An answer to this is provided by the following theorem:

**Theorem 9.2** *Let PM be a realizable plane model of genus = 0 (i.e., a plane model topologically equivalent to a sphere). Then the polygon pasting and vertex joining operations are capable of reducing PM to a plane model that has just one vertex and one polygon, but no edges.*

**Proof:** *Suppose there is an edge E of PM such that it cannot be removed with either a polygon pasting or a vertex joining operation. Therefore, E must occur twice at the same polygon, and twice at the same vertex. Hence, E forms a closed curve on the 2-manifold M modeled by PM that does not split M into two components, and PM must be of genus* $> 0$.[1] *By contradiction, no such E can occur, and all edges of PM can be removed.*

So, by means of the pasting and joining operations only, *all* plane models of the sphere can be reduced to a one-vertex, one-polygon model. We call this special model the *skeletal plane model.* A sample case of the theorem is presented in Figure 9.2. In this particular case, all vertices of the object are first made to occur in a single polygon by removing edges with the pasting operations until no further faces can be merged (Figure 9.2(a,b)). Thereafter, all remaining edges are removed with the joining operation until only the skeletal plane model (Figure 9.2(c)) remains.

Of course, the skeletal plane model as such is not a particularly useful model of a sphere or a cube. Its significance for our purposes is due to the following:

**Corollary 9.3** *Let PM be a realizable plane model of genus 0. Starting from a plane model consisting of a single vertex and a single polygon, PM*

[1] See discussion on the first Betti number $h_1$ on page 45.

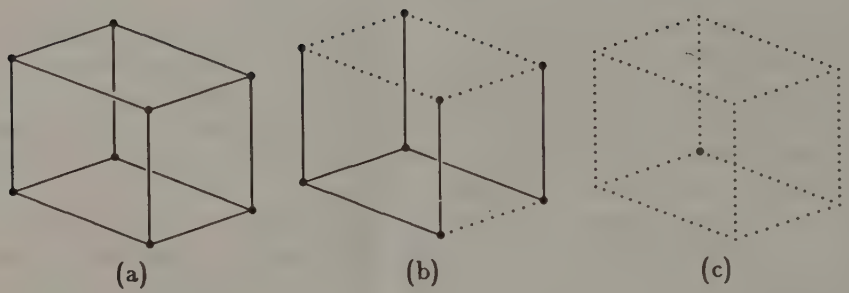



**Figure 9.2** Reducing a cube to the skeletal plane model.

*can be created by a sequence of e polygon cutting or vertex splitting operations, where e is the number of edges of PM.*

**Proof:** *By Theorem 9.2, all edges of PM can be removed with a sequence S consisting of e pasting and joining operations. Because the polygon cutting and the vertex splitting operations are the exact inverses of the pasting and joining operations, the sequence $S^{-1}$ consisting of the cutting and splitting operations corresponding with the operations of S in reverse order is a sequence satisfying the claim.*

That is, starting from the skeletal plane model, *all* plane models of genus 0 can be created by means of the cutting and splitting operations! Therefore, in order to create an arbitrary plane model of genus 0, we only need to supply the "prototype" skeletal plane model as a primitive, and work on it with the two operations.

As the two manipulation operations do not alter the global topological properties of the plane model they operate on, we call them *local topological operations.*

### 9.1.2 Global Topological Operations

In addition to the local topological operations, we need manipulation operations which are capable of producing objects whose genus is greater than zero. As these operations must definitely alter the global topological properties of a plane model, we call them *global topological operations.*

At this point a slight diversion back to the topology of surfaces is necessary. Given two surfaces, it is always possible to construct a new surface by cutting a small disk off from each original surface, and pasting the two

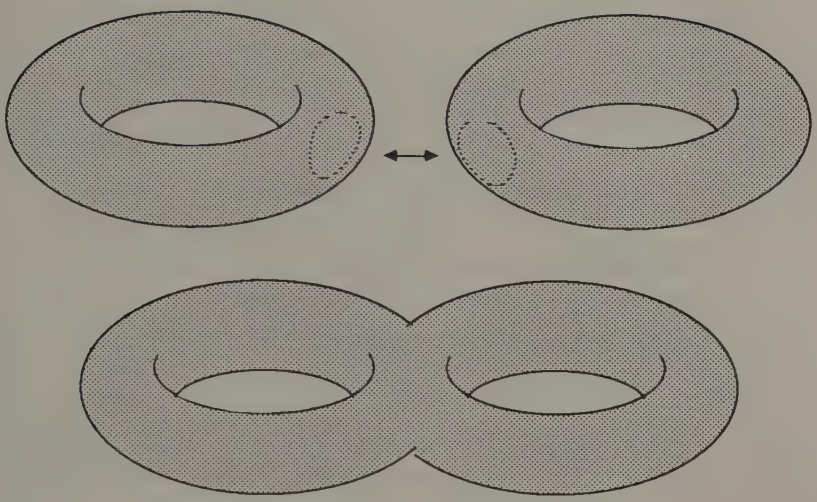


**Figure 9.3** Connected sum of two tori.

surfaces together along the boundary of the cutouts. For instance, the *two-holed torus* can be created in this fashion from two tori (see Figure 9.3).

The surface resulting from the cut-and-paste procedure is called the *connected sum* of the two original surfaces. That the cut-and-paste procedure is general enough for creating all surfaces we are interested in, is shown by the so-called *classification theorem*:

**Theorem 9.4** *Every orientable surface is topologically equivalent either to the sphere, or the connected sum of n tori.*

The proof of the classification theorem is beyond the scope of this text; again, see [52] for a good exposition.

The cut-and-paste procedure can be readily interpreted in the language of plane models. In the interpretation, the "cutting of disks" becomes the creation of two polygons in the two plane models at the desired positions. The "pasting" is modeled as an operation that combines those two polygons with a new edge that will occur twice at the new combined polygon.

For instance, Figure 9.4 depicts the plane models of a cube and a tetrahedron. They can be combined by joining the polygon $p$ of the cube and the "external" polygon with a new edge. Observe that new edge occurs twice in the combined polygon $p'$; i.e., $p'$ now has nine edges.

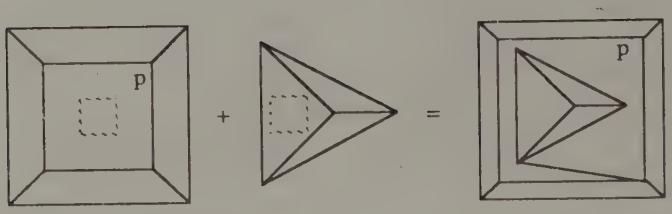



**Figure 9.4** Connected sum of two plane models.

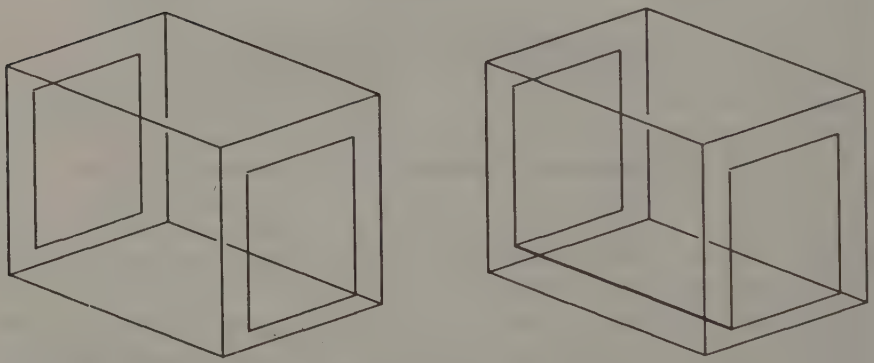


**Figure 9.5** Connected sum on a single plane model.

If we interpret the two original plane models of Figure 9.4 as a single disconnected plane model, we may observe that the connected sum operation alters the connectivity of the model. By extending the use of the connected sum operation to two polygons of a connected (component of) plane model, we can also alter the genus of a plane model, as desired. For instance, we can produce a hole through the cube of Figure 9.5 by joining the two polygons at its ends with a new edge.

Again, the connected sum operation has a natural inverse operation. This *connected minus* operation can be used to remove an edge occurring twice at a polygon. As a result, the polygon is divided into two polygons.

Furnished with the connected sum operation and its inverse, we can show the following:

**Theorem 9.5** *Let PM be a realizable plane model. Using the local topological operations (polygon cutting and pasting, vertex splitting and joining), and the connected sum and connected minus operations, all edges of PM can be removed, i.e., PM can be reduced to the skeletal plane model.*

**Proof:** *Consider an arbitrary edge E of PM. The following case analysis applies:*

1. *E separates two distinct polygons. In this case, E can be removed with the polygon pasting operation.*
2. *E occurs in a single polygon between two distinct vertices. In this case, E can be removed with the vertex joining operation.*
3. *Otherwise, E can be removed with the connected minus operation.*

*Because all cases are covered, all edges can be removed.*

Similarly to the corresponding results of Theorem 7.2 and Corollary 7.3, the following corollary is readily seen to be valid:

**Corollary 9.6** *Let PM be a realizable plane model. Starting from the skeletal plane model, PM can be created by a finite sequence of local topological operations and connected sum or minus operations.*

**Proof:** *Analogous to Corollary 9.3.*

That is, *all* realizable plane models can be created with the local topological operations and the connected sum and minus operations.

### 9.1.3 Soundness of Plane Model Operations

Corollary 9.6 tells us that every plane model of interest to us (that is, all realizable plane models) can be generated by means of the plane model manipulation operations. Therefore, the operations are *complete*, i.e., they are powerful enough for describing all realizable plane models.

It is natural to ask whether the operations also are *sound*, i.e., whether we can be certain that no others besides the realizable plane models can be generated through their use. That this really is so is shown by the following theorem:

**Theorem 9.7** *Both the local plane model manipulation operations (polygon cutting and pasting, vertex splitting and joining), and the global plane model manipulation operations (connected sum and connected minus) are* sound, *i.e., they cannot create nonrealizable plane models.*

**Proof:** *Let us consider an arbitrary finite sequence of plane model manipulation operations. The proof will be inductive on the length of the sequence.*

*First, the criteria for realizable plane models as given by the Definitions 8.9 and 8.10 of Chapter 8 are as follows:*

1. Edge identification: *Every edge of the plane model is topologically identified with exactly one other edge.*
2. Cyclical identification: *The polygons identified at each vertex can be arranged in a cycle such that each consecutive pair of polygons in the cycle is identified at an edge adjacent to the vertex.*
3. Orientability: *The polygons of the model can be oriented so that for each pair of identified edges, one edge occurs in its positive orientation in the direction of its polygon, and the other in the negative orientation.*

*As the basis of the induction, the skeletal plane model trivially satisfies the three conditions. From there on, we have exactly six cases corresponding to the six possible operations to consider. In each case, we shall have to show that the operation will preserve the criteria 1–3 stated above.*

*Let us only elaborate two of the cases here and leave the rest as an exercise to the reader. First, the vertex splitting operation adds a pair of identified edges that occur in opposite orientations in their respective polygons in the plane model. Hence, if the plane model originally satisfies the edge identification and the orientability constraints, it will do so also after the operation. As for the cyclical identification constraint, observe that the vertex splitting operation divides the cycle of edges of a vertex into two sequences; hence also cyclical identification will remain unaltered.*

*As vertex splitting and face cutting are duals of each other, the reasoning above immediately has the consequence that the face splitting operation will not alter the validity criteria either. Finally, also the connected sum operation can be shown to preserve the criteria.*

*Because none of the cases violates the inductive hypothesis, an arbitrary sequence of the operations will create a model that satisfies the realizability criteria, and we have the theorem.*

Together, Corollary 9.6 and Theorem 9.7 state that the local and global manipulations are *sound and complete* for the family of realizable plane models. Therefore, they fill exactly the desired characteristics stated in the start of this section.

## 9.2 MANIPULATION OF BOUNDARY MODELS

As seen in the previous section, a small collection of manipulation operations (and their inverses) are sufficient for describing all plane models of

interest. According to the theory, only three basic types of manipulative operations were needed:

1. A "prototype" primitive that creates skeletal plane models.

2. Two local topological operations that subdivide the sequences of edges of a face or of a vertex.

3. A global topological operation that implements a connected sum of two polygons (either from distinct models or one model).

The theory carries over to boundary models. The manipulation operations on plane models will have their counterparts in the form of *Euler operators* that act on the topology of a boundary model data structure. Being realizations of the theoretical plane model operations for the domain of real bounary data structures, Euler operators share their completeness and soundness stated as Corollary 9.6 and Theorem 9.7.

### 9.2.1 Notation and Conventions

Euler operators were originally introduced by Baumgart [13,14] in the context of the winged-edge data structure. As this data structure is the natural basis for discussing the operators, we shall assume the use of the data structure of Figure 6.6, extended with loop nodes for representing nonsimple faces. For convenience of language, the internal loops will be termed *rings*. We shall also use the term *shell* introduced in Section 3.5.6 to denote a connected component of a boundary data structure.

We shall also use the convention of having *empty loops* in the data structure, i.e., loops consisting of just a single "lone" vertex with no edges at all. Such entities are useful, for instance, for representing the boundary model counterpart of the skeletal plane model.

By historical convention, the operators are usually denoted by rather cryptic, mnemonic names. The key to the names to be used in this text is given in the below:

| | | |
|---|---|---|
| M — make | V — vertex | H — hole |
| K — kill | E — edge | R — ring |
| S — split | F — face | |
| J — join | S — solid | |

For instance, the name "mev" translates as "Make Edge, Vertex".

It should be borne in mind that Euler operators can be implemented in several ways on top of widely varying data structures. We shall emphasize this by using the SMALL CAPS font when refering to an Euler operator

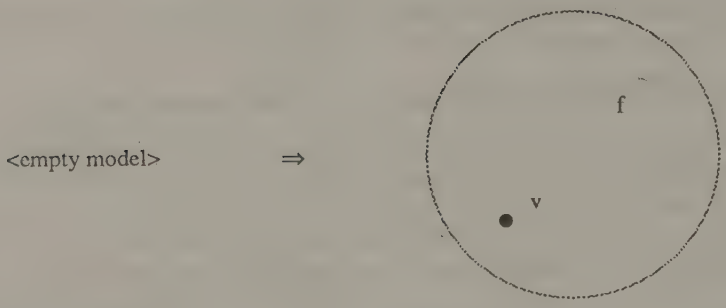



**Figure 9.6** The MVFS operator.

"in general," without regard for a particular implementation. This is in contrast to our usual convention of using the `typewriter` font when refering to an actual piece of code.

### 9.2.2 Skeletal Primitives: MVFS and KVFS

The theory tells us that by means of local and global topological manipulations, all plane models of interest can be created from a single skeletal plane model. This result means that a single skeletal primitive is sufficient for us.

The operator MVFS takes this role in our collection. As the name suggests, it creates from scratch an instance of the data structure that has just one face and one "lone" vertex. Hence the new face has one "empty" loop with no edges at all. The effect of MVFS is depicted in Figure 9.6.

The "solid" created by MVFS may not satisfy the intuitive notion of a solid object. Nevertheless, it it useful as the initial state of creating a boundary model with a sequence of Euler operators.

Similarly to the plane model manipulation operations, all Euler operators will have corresponding inverse operators that can undo the effect of the "positive" operator. The inverse of MVFS is called KVFS, and it destroys a skeletal instance of the data structure identical to that created by MVFS.

### 9.2.3 Local Manipulations

In analogy to local topological operations on plane models, we need operators that can work on the local topological properties of a boundary model.

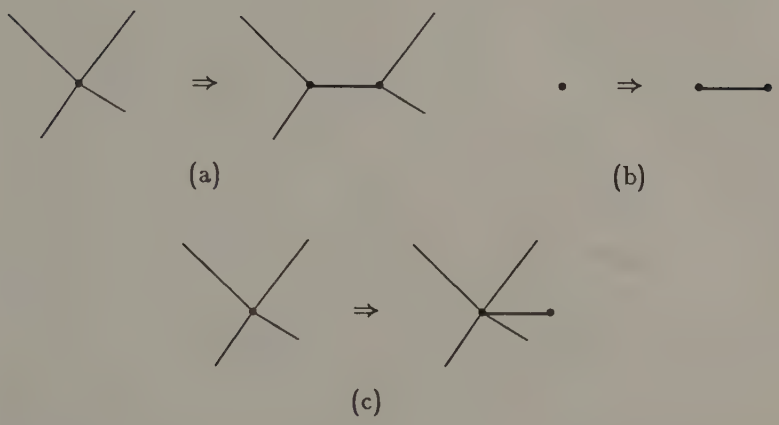



**Figure 9.7** The MEV operator.

## MEV, KEV

Operators MEV and MEF (to be described in the next section), with their inverses (KEV, KEF) correspond exactly with the polygon and vertex splitting operations for plane models. Hence MEV subdivides the cycle of edges of a vertex into two cycles by "splitting" a vertex into two vertices, joined with a new edge (see Figure 9.7(a)). The net effect of this is to add one vertex and one edge in the data structure, as suggested by the name of the operator.

The applicability of MEV is extended to "lone" vertices (i.e., vertices appearing in empty loops such as the one created by MVFS) by considering the result of subdividing a vertex with no edges at all as consisting of two vertices joined with an edge (see Figure 9.7(b)). We also include the case that a new vertex joined with the old vertex by means of a new edge is created as depicted in Figure 9.7(c).

The inverse operator KEV is capable of undoing any of the three cases of Figure 9.7. In other words, given an edge connecting two distinct vertices, KEV can remove the edge, collapse the vertices into one and merge their edge cycles.

## MEF, KEF

The operator MEF subdivides a loop by joining two vertices with a new edge. Its net effect is to add one new edge and face into the data structure.

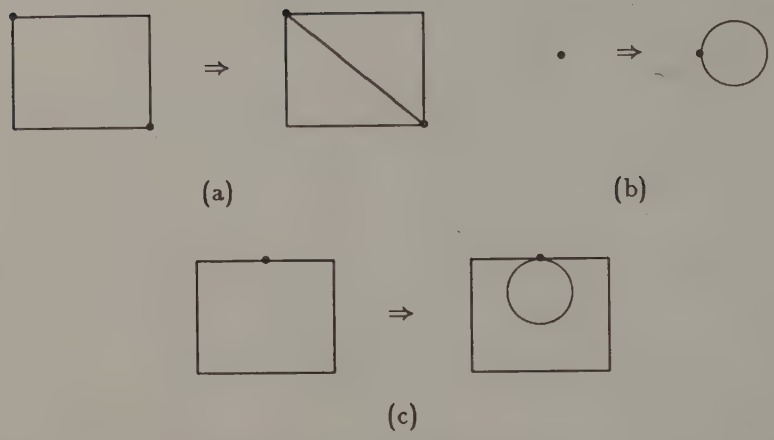



**Figure 9.8** The MEF operator.

In addition to the ordinary case of Figure 9.8(a), we also extend the applicability of MEF to empty loops in strict analogy with MEV. Hence the result of subdividing a "lone" vertex consists of a "circular" edge separating two faces. Observe that the starting and final vertices of such an edge are equal; this is a case admitted by the winged-edge data structure (Figure 9.8(b)). As a more general case of this, it is always possible to "connect" a vertex to itself by means of MEF as depicted in Figure 9.8(c).

Again, the inverse operator KEF can undo the effect of MEF in each case. More accurately, given an edge adjacent to two distinct faces, KEF is capable of removing the edge, and joining the two faces into one whose bounding loop is the result of merging the two original boundaries.

The reader is urged to contemplate Figures 9.7 and 9.8 until their analogy is clear to him. Note also how the descriptions of KEV and KEF are "duals" to each other.

## KEMR, MEKR

Operators MEV and MEF are "fundamental" in that their significance can be derived directly from the theory of plane models. In contrast, the remaining local manipulation is a "convenience" operator whose necessity is due to our conventions rather than the underlying theory.

In the above, "empty loops" were used to realize the skeletal plane mode in the boundary data sturcture domain. To make full use of empty loops, it is useful to introduce an operator specially tailored for their creation.

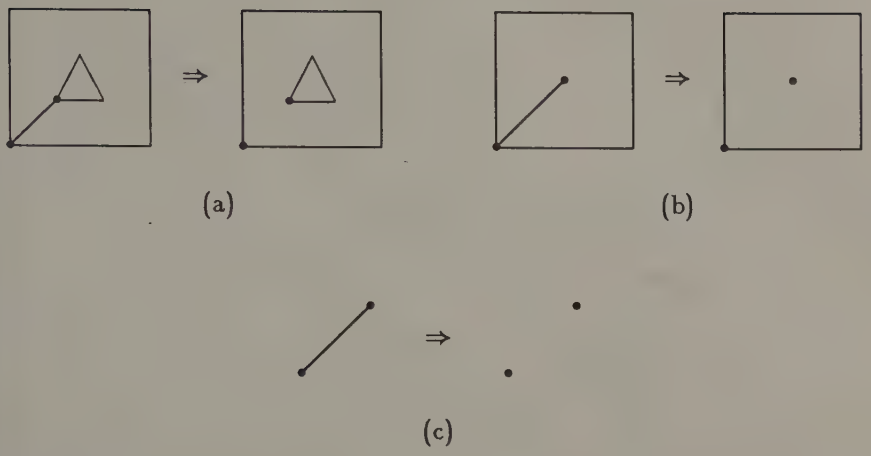



**Figure 9.9** The KEMR operator.

The resulting operator KEMR splits a loop into two new ones by removing an edge that appears twice in it (see Figure 9.9(a)). Hence KEMR divides a connected bounding curve of a face into two bounding curves, and has the net effect of removing one edge and adding one ring to the data structure. The special cases that one or both of the resulting loops are empty are also included (Figure 9.9(b, c)).

The inverse operator MEKR can merge two loops of a face by joining one vertex of each with a new edge.

### 9.2.4 Global Manipulations: KFMRH, MFKRH

None of the operators discussed so far is capable of modifying the global topological properties of the data structure e.g. by dividing a solid into two components or by creating a "hole." This is the task of the remaining Euler operator.

Operator KFMRH is a realization of the connected sum operation discussed in Section 9.1.2. Given two faces (say, $f_1$ and $f_2$), it joins them into one face by transforming the bounding loop of $f_2$ to a ring of $f_1$. Hence its net effect is to remove one face ($f_2$) and add one ring. Note that KFMRH has no effect on the local arrangement of edges and vertices of its argument faces—it is truly a global manipulation. This also means that it is hard to illustrate its effect. One attempt is given in Figure 9.10(a).

KFMRH is actually a misnomer, because the operator does not neces-

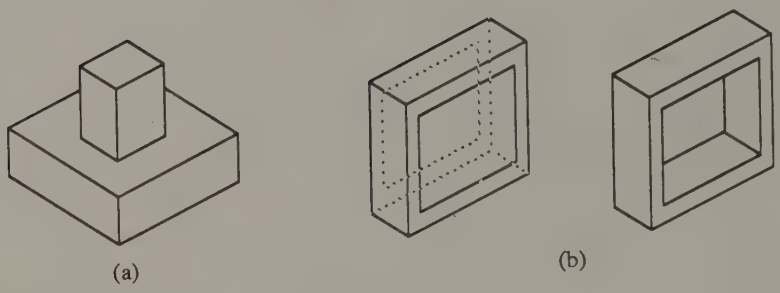



**Figure 9.10** The KFMRH operator.

sarily create a "hole". Actually, KFMRH creates a hole only if the two argument faces belong to the same shell. When applied to faces belonging to two distinct shells, its effect is to combine them into one shell, and the name "KFSMR" (where S now stands for "shell") would seem a better condensed description of what really takes place (see Figure 9.10(b)). But neither of the two names is better than the other, and we shall stick to KFMRH.

Again, an inverse operator MFKRH is capable of reversing the effect of KFMRH. It modifies a ring of a face as the bounding loop of a new face.

Recall that the connected sum operation for plane models was specified to add a new edge connecting the two polygons to be joined, while KFMRH achieves the same effect by making the boundary of one face a ring in the other. Otherwise, the operations are strictly analogous.

## 9.3 AN EXAMPLE OF EULER OPERATORS

Let us clarify the use of Euler operators by means of a simple example, the construction of a rectilinear box with one rectilinear hole. The construction is depicted in Figure 9.11 in terms of both plane models and three-dimensional space models.

The construction starts by creating the skeletal model by means of a MVFS (a). By applying MEV three times, three edges and vertices are added so as to create (b). By joining the end vertices of the edge string (b) with a MEF, the model (c) is created. Note that the space model of (c) consists of two planar faces tightly upon each other like the two sides of a piece of paper. Such a model is called a *lamina*.

Four MEV's lead us to the model (d) that has the side edges of the emerging box. Their tips are joined with four MEF's to form the side faces

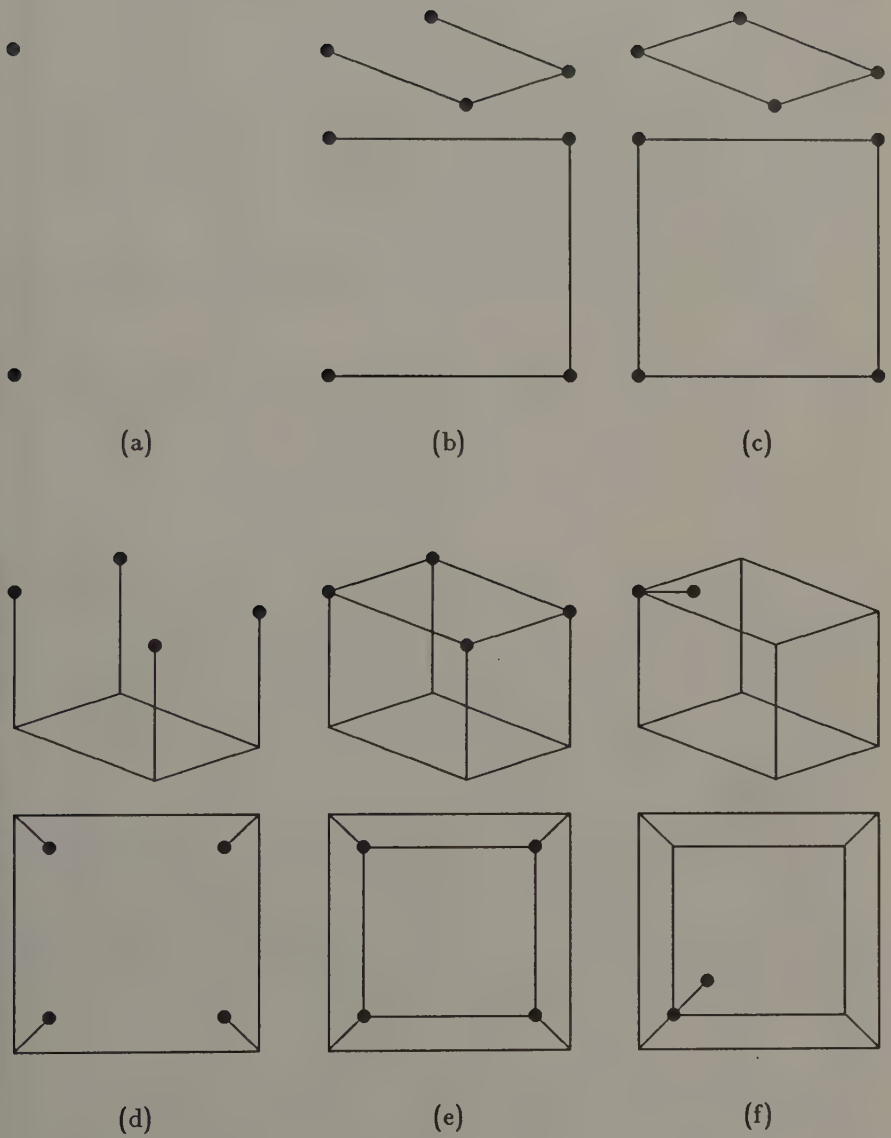



**Figure 9.11** Example of Euler operators.

(e). We now have formed the model of a simple box.

To create the hole, we first need to form an interior face on the top of the box (e). We do that by adding an edge on the top face with a MEV (f), and removing the edge with a KEMR (g). This gives us an empty loop with one vertex only inside the top face. After three additional edges are drawn from that vertex with MEV's (h), the interior face can be formed with a MEF (i). Now the side faces of the hole can be formed with MEV's (j) and MEF's (k). Steps (h) through (k) are strictly analogous to (b) through (e).

We now have formed all edges and vertices of the desired result. To finish the description, the bottom face of the emerging hole is "subtracted" from the bottom face of the box by means of a KFMRH (l). This is the same operation as that depicted in Figure 9.10(a).

Of course, to actually create the box, we need to assign coordinates to vertices as they are created. This can be accomplished by operators that create vertices (i.e., MVFS and MEV).

## 9.4 PROPERTIES OF EULER OPERATORS

### 9.4.1 Euler-Poincaré Formula Revisited

Recall the Euler-Poincaré Formula 3.2 of Section 3.5.6:

$$v - e + f = 2(s - h), \tag{9.1}$$

where $v$, $e$, $f$, $s$, and $h$ denote the numbers of vertices, edges, faces, shells, and holes in a solid.

From the theory of plane models it it clear that a collection of faces, edges, and vertices can be a valid boundary model only if the numbers of these elements satisfy Equation 9.1. Hence it can be considered a necessary integrity criterion for boundary models.

Equation 9.1 can be modified to make it consistent with our boundary data structure conventions by introducing a new parameter $r$, standing for the number of rings. Obviously, each ring can be removed by introducing a new edge that connects it into the external bounding loop of its face (see Figure 6.7). No new vertices or faces are added in this operation, but the number of edges after the removal of $r$ rings will be $e' = e + r$. Substituting $e'$ for $e$ in Equation 9.1 gives us the desired form

$$v - e + f = 2(s - h) + r \tag{9.2}$$

that describes a necessary condition for the numbers of elements of a boundary data structure ($v$, $e$, $f$, $r$), and the global characteristics of the solid modeled ($s$, $h$).

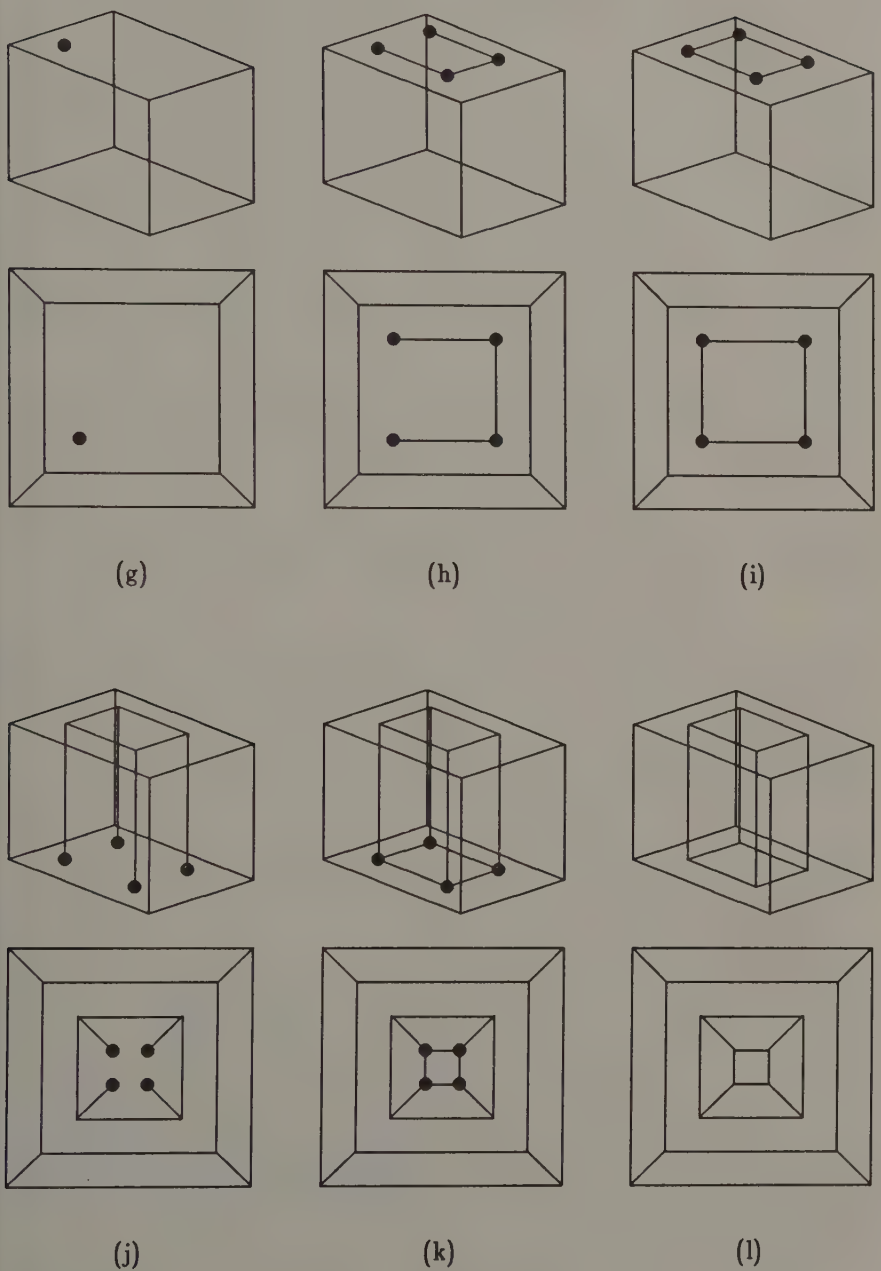



**Figure 9.11** Example of Euler operators (cont.).

### 9.4.2 Algebraic Properties of Euler Operators

Observe that the data structure created by MVFS has $v = f = s = 1$ and $e = h = r = 0$. By substituting these numbers into Equation 9.2, we can see that they make it hold. Furthermore, all other operators will add or remove entities of the data structure in a fashion that keeps the condition expressed by Equation 9.2 satisfied. For instance, MFKRH adds one face and removes a shell and a ring. In terms of Equation 9.2, its effect is to replace $f$ by $f' = f + 1$, $r$ by $r' = r - 1$, and $h$ by $h' = h - 1$. By substituting $f'$, $r'$, and $h'$ into 9.2 one can readily see that the formula will remain satisfied. Hence Euler operators have the property that *the numbers of elements in each data structure created by them will satisfy the Euler formula of Equation* 9.2.

This observation suggests that Euler operators could be analyzed in purely algebraic terms. Following Braid et al. [19], let us consider a six-dimensional discrete space, and call its axes $v$, $e$, $f$, $h$, $r$, and $s$. Then Equation 9.2 can be interpreted as the equation of a five-dimensional hyperplane $E$ of the space. In this view, the analogy of a boundary model is a point

$$P = (v\ e\ f\ h\ r\ s),$$

of the six-dimensional space and the condition of Equation 9.2 can be interpreted as stating that "interesting" points of the space are those lying on the plane $E$.

The hyperplane $E$ is itself a five-dimensional discrete subspace of the six-dimensional space. Therefore, it can be spanned by five six-vectors $V_i$ satisfying the following conditions:

1. Each vector $V_i$ lies on $E$.

2. Vectors $V_i$ are linearly independent.

After such a set of base vectors of $E$ has been chosen, all points of $E$ can obviously be expressed as linear combinations of the base vectors.

In the analogy suggested above, Euler operators take the role of the base vectors. In this view, our collection would be interpreted as the set of base vectors summarized in Table 9.1. Obviously, an infinite amount of other sets of base vectors are possible, and every set can be interpreted as a particular collection of Euler operators. In practice, it is desirable to choose simple operators that correspond with "short" vectors of $E$, and the published collections of Euler operators [14,33,34,19,4,78,75] are almost identical with each other.

This algebraic view of Euler operators can be applied to gain interesting insight into their properties. To see how, let us arrange the base vectors

| *Operator* | *Base Vector* | | | | | |
|---|---|---|---|---|---|---|
| | *v* | *e* | *f* | *h* | *r* | *s* |
| MEV | 1 | 1 | 0 | 0 | 0 | 0 |
| MEF | 0 | 1 | 1 | 0 | 0 | 0 |
| MVFS | 1 | 0 | 1 | 0 | 0 | 1 |
| KEMR | 0 | −1 | 0 | 0 | 1 | 0 |
| KFMRH | 0 | 0 | −1 | 1 | 1 | 0 |
| KEV | −1 | −1 | 0 | 0 | 0 | 0 |
| KEF | 0 | −1 | −1 | 0 | 0 | 0 |
| KVFS | −1 | 0 | −1 | 0 | 0 | −1 |
| MEKR | 0 | 1 | 0 | 0 | −1 | 0 |
| MFKRH | 0 | 0 | 1 | −1 | −1 | 0 |

**Table 9.1** Euler operators.

corresponding with the Euler operators of our collection, together with the coefficients of Equation 9.2 into the matrix $M$ given below:

$$M = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 1 & -1 & 1 & 2 & -1 & -2 \end{bmatrix} \tag{9.3}$$

Observe that the coefficient vector on the last row of $M$ acts as the normal vector of the hyperplane $E$. Because Euler operators form a base of $E$, the matrix has an inverse. The inverse matrix $M^{-1}$ is given below.

$$M^{-1} = \frac{1}{12} \begin{bmatrix} 9 & -5 & 2 & -2 & 3 & 1 \\ -3 & 5 & -2 & 2 & -3 & -1 \\ 3 & 7 & 2 & -2 & 3 & 1 \\ -6 & 2 & 4 & 8 & -6 & 2 \\ 3 & 5 & -2 & 2 & 9 & -1 \\ -6 & -2 & 8 & 4 & -6 & -2 \end{bmatrix} \tag{9.4}$$

The inverse matrix can be used to calculate the "Euler coordinates" of any $[v\ e\ f\ h\ r\ s]$-tuple. For instance, the object constructed in Figure 9.11 has $v = 16$ vertices, $e = 24$ edges, $f = 10$ faces, $r = 2$ rings, $h = 1$ hole, and $s = 1$ shell. Hence it corresponds with the location

$$P = [16\ 24\ 10\ 1\ 2\ 1]$$

of the discrete six-dimensional space. Now the product

$$P \times M^{-1} = [15\ 10\ 1\ 1\ 1\ 0]^T$$

tells us that to model this solid, a total of 15 MEV's, 10 MEF's, 1 MVFS's, 1 KFMRH's and 1 KEMR are needed at the minimum. That the last component is zero shows that the original 6-tuple satisfies Equation 9.2. Observe that the construction of Figure 9.11 uses exactly these numbers of operators; hence, it it "optimal" in the length of the construction.

The algebraic analysis gives us hence a mental tool for discussing the complexity of Euler operator descriptions of boundary models. As another example of this, if $s = 1$, the general product

$$[v\ e\ f\ h\ r\ s] \times M^{-1}$$

yields the following operator counts:

$$\begin{array}{ll} v-1 & \text{MEV's} \\ f+h-1 & \text{MEF's} \\ 1 & \text{MVFS} \\ h & \text{KFMRH's} \\ r-h & \text{KEMR's} \end{array} \qquad (9.5)$$

Clearly, Equations 9.5 give the smallest numbers of operators needed to model an object corresponding with location $[v\ e\ f\ h\ r\ 1]$ of the six-dimensional discrete space.

Unfortunately, the inverse operators count as $-1$ in Equations 9.5, which makes their interpretation difficult. Nevertheless, because all models of interest have $v, f > 0$ and $f > h$, the four first counts are nonnegative. This suggests that all connected objects can be modeled without the "inverse" operators KEV, KEF, KVFS, and MFKRH, a fact that will be shown in Chapter 16. Also, from the fact that the last line can be negative, we can infer that some models cannot be created without the MEKR operator.

### 9.4.3 Descriptive Power of Euler Operators

Euler operators remove some part of the complexity of dealing with boundary data structures. Does this impose a penalty as far as the generality and flexibility is concerned? Is it possible to create arbitrary models under restriction that only Euler operators are used in their construction?

The algebraic view to Euler operators presented in the last section gives us some insight into these questions. Obviously, each 6-tuple $[v\ e\ f\ h\ r\ s]$ can be associated with a tuple of "Euler coordinates" through the inverse of the transition matrix $M$. But there is still some concern, however. As the

reader can easily verify with a counterexample, a 6-tuple does not uniquely determine a solid. Can we produce all objects that have the same tuple? Furthermore, while the discussion of the last section suggests that the number of operators needed to model a solid grows linearly with the complexity of the object, it is not completely clear whether an operator sequence of that length exists for all objects, because inverse Euler operators contribute as $-1$ to the counts given by the inverse transition matrix.

The theory of plane model manipulations presented in Section 9.1 resolves the first question. In particular, Theorems 9.5 and 9.7, and Corollary 9.6 can be reinterpreted in the domain of boundary data structures and Euler operators. In this new form, the theorems are as follows:

**Theorem 9.8** *Let $S$ be a valid boundary data structure (i.e., $S$ satisfies all topological integrity constraints). Then there exists a finite sequence of Euler operators that can completely remove $S$.*

**Corollary 9.9** *All valid boundary data structures can be created with a finite sequence of Euler operators.*

The direct proofs are analogous to those of Theorem 9.5 and Corollary 9.6. In Chapter 16, we shall give a constructive proof by developing an algorithm that actually constructs such a sequence of operators.

**Theorem 9.10** *Euler operators are* sound, *i.e., they cannot create topologically invalid boundary data structures.*

Again, the direct proof is similar to that of Theorem 9.7. As an additional case, the soundness of KEMR and MEKR would have to be demonstrated.

Two notes on the soundness theorem may be needed. First, to model anything at all, a sequence of operators must satisfy certain "syntactic" validity criteria pertaining to a particular way of representing Euler operators. For instance, all entities that operators work on must exist and be of proper type. Say, the operator KEF can be applied only to an edge that appears in two distinct faces.

Second, the theorem has nothing to say on the *geometric* validity of the resulting model; by assigning inappropriate geometric information to faces, edges, and vertices, it is still perfectly possible to create invalid models that have no physical significance.

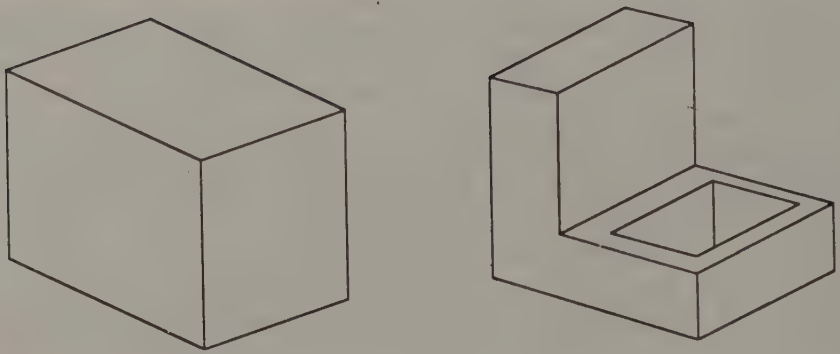


**Figure 9.12** Sample solids.

## PROBLEMS

9.1. Complete the proof of Theorem 9.7 on page 145.

9.2. Give an Euler operator construction in the style of Figure 9.11 for the following objects (see Figure 9.12):

(a) A rectilinear block with six faces and eight vertices.

(b) An "L-brick with hole" with 12 faces and 20 vertices.

9.3. Show that some solids cannot be described without the MEKR operator.

*Hint:* Equations 9.5 suggest that for such an object, $h > r$.

9.4. Provide an example of two different boundary models having the same $[v\ e\ f\ h\ r\ s]$-tuple.

9.5. Demonstrate the soundness of the Euler operators KEMR and MEKR.

*Hint:* You will have to show that they preserve the conditions of edge identification, cyclical identification, and orientability.

## BIBLIOGRAPHIC NOTES

The idea of using plane models to argue rigorously about the formal properties of Euler operators was originally presented in the article [75] of the author. Most of the material presented in this chapter stems from this publication, but appears here in extended and clarified format.

The discussion on algebraic properties of Euler operators presented in Section 9.5.2 is based on the exposition of Braid et al. [19] with some minor changes in details.

# Chapter 10

# HALF-EDGE DATA STRUCTURE

This chapter describes the *half-edge data structure* that forms the main solid representation of GWB.

## 10.1 REQUIREMENTS

Chapters 3 and 9 use plane models to discuss properties of surfaces and operations on them in a mathematically rigorous fashion. In the design of GWB, our aim will be to implement this methodology in practice. In particular, we need a data representation that can conveniently support the implementation of Euler operators.

Plane models and their manipulation operations are abstract mathematical entities, and hence very general. In particular, they include non-intuitive models (such as the skeletal plane model) that are necessary for being able to record all intermediate steps of a model description sequence.

Figure 10.1 gives some other unintuitive plane models. In particular, case (a) depicts an edge that occurs twice at a face and case (b) a polygon with just one "circular" edge whose start and end vertices are equal. As unintuitive as they are, these plane models nevertheless may act as intermediate steps during the description of plane models (c) and (d). In fact, without permitting these kinds of cases, the manipulation operations of plane models would be of quite limited use.

The inclusion all special cases becomes yet more important if we shift our attention from the domain of plane models to the domain of boundary data structures. Geometric models are not static entities that only

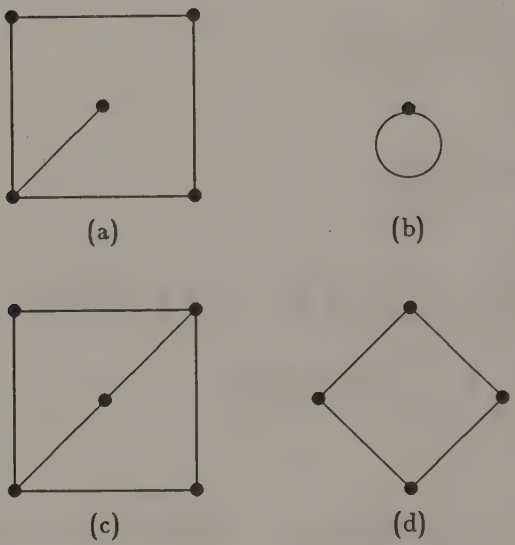



**Figure 10.1** Special cases of plane models.

represent information, but data that must be worked on. For instance, a "drawing"-oriented user interface must be capable of adding new polygons, edges, and vertices to a boundary model, and it is natural that special cases will occur. Similarly, an algorithm for Boolean set operations may need to create quite unintuitive intermediate models when it manipulates boundary data structures so as to compute the desired result. Hence, all "special cases" of Figure 10.1 apply also for boundary models.

A practical boundary data structure must, of course, include also the geometric information necessary for describing a mapping from the plane model to a surface. As we shall be dealing with polyhedral models only, this mapping will be represented simply in terms of coordinates assigned to vertices. Later in Chapter 18 we shall discuss extensions to this.

## 10.2 OVERVIEW OF THE DATA STRUCTURE

According to Definition 3.8 on page 40, a plane model is a planar directed graph $\{N, A, R\}$ of vertices $N$, edges $A$, and polygons $R$. On top of the planar graph, a special topology is defined by identifying edges and vertices of individual polygons with each other. The identification allows the representation of nonplanar models.

Any data structure that aims at representing the similar information must be capable of representing both the graph structure and the identification structure of a plane model. Fortunately, according to Definition 3.9, page 41, we know that the identification structure of a realizable plane model is very simple: each edge is identified with exactly one other edge, and the polygons identified at each vertex form a single cycle. These features can be exploited in the design of the data structure.

The data structure we shall use is a variation of the full winged-edge data structure of Figure 6.7, page 109; we shall call it the *half-edge data structure*. Before going into the details of its implementation, we give first an overview to its features and its relationship to plane models in the next sections.

## 10.2.1 Graph Representation

For representing the graph $\{N, A, R\}$ we shall use a five-level hierarchic data structure, consisting of nodes of type *Solid*, *Face*, *Loop*, *HalfEdge*, and *Vertex*.[1] The hierarchic features of the node types are described below; the next section describes additional information that represents the identification.

### Node Description

**Solid** Node *Solid* forms the root node of an instance of the half-edge data structure. At any point of time, several instances of the data structure may be in existence; to access any of them, a pointer to its solid node is needed. The solid node gives access to faces, edges, and vertices of the model through pointers to three doubly-linked lists. (Edge nodes are presented in the next section.) All solids are linked into a doubly-linked list realized by means of pointers to the next and the previous solid of the list.

**Face** Node *Face* represents one planar face of the polyhedron represented by the half-edge data structure. In GWB, a "face" is defined as a planar polygon whose interior is connected. We choose to include the convenience of faces with multiple boundaries into the data structure; hence each face is associated with a list of loops, each representing one polygonal boundary curve of the face.

As all faces represent planar polygons, one of the loops can be designated as the "outer" boundary, while others represent the "holes" of

[1] Even at the risk of confusion, we shall have "vertices" and "edges" both in the plane models and in the half-edge data structure. Which is meant should be clear from the context. The reason for the term "half-edge" is explained in due course.

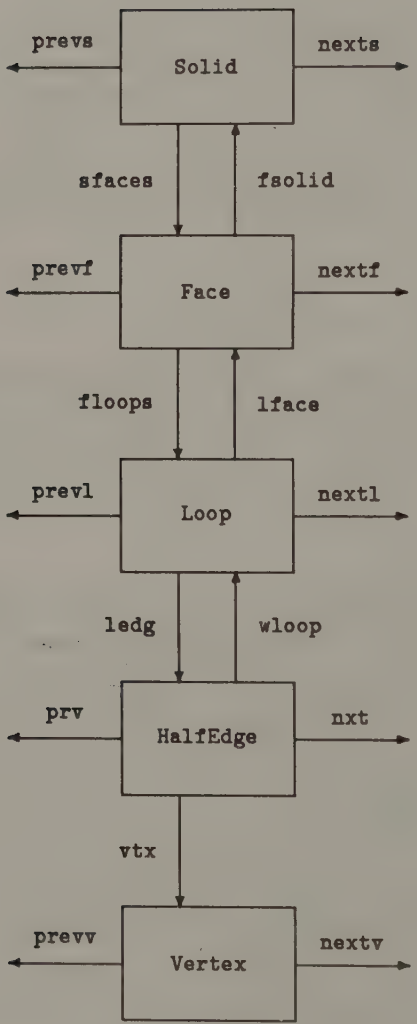



**Figure 10.2** Hierarchic view of the half-edge data structure.

a face. This is realized in terms of two pointers to a loop node, one that points to the "outer" boundary, and one that points to the first loop in the doubly-linked list consisting of all loops of the face. We shall follow the convention of calling hole-loops *rings*.

Faces have a vector of four floating-point numbers that represent its plane equation. To realize the doubly-linked list of all faces of a solid, each face includes pointers to the previous and the next face in the list. Finally, each face has a pointer to its parent solid.

**Loop** Node *Loop* describes one connected boundary as discussed above. It has a pointer to its parent face, a pointer to one of the halfedges that form the boundary, and pointers to the next loop and to the previous loop of the face.

**HalfEdge** Node *HalfEdge* describes one line segment of a loop. It consists of a pointer to its parent loop and a pointer to the starting vertex of the line segment in the direction of the loop. Pointers to the previous and the next halfedge realize a doubly-linked list of halfedges of a loop; hence the final vertex of the line segment is available as the starting vertex of the next halfedge.

**Vertex** Node *Vertex* contains a vector of four floating-point numbers that represent a point of $E^3$ in the homogeneous coordinate representation explained in Appendix A. Two pointers to the next and the previous vertex realize a doubly-linked list of vertices of a solid.

The hierarchy is depicted in Figure 10.2, including the C names of some of the pointers.

### 10.2.2 Representation of Identification

The strict hierarchy of Figure 10.2 does not indicate directly how the individual faces are related to each other, except when several polygons refer to the same vertex node. We shall add one additional node type that makes the face-to-face relationships explicit by representing the identification of their line segments.

Recall that every edge of a valid plane model is identified with exactly one other edge. Hence, each halfedge should be associated with exactly one other halfedge. We shall use the additional node type *Edge* to record this information. As described below, we also add a few data items to certain nodes of the hierarchy to fully exploit the identification information.

**Edge** Node *Edge* associates two halfedges with each other; intuitively, is combines the two halves of a full edge together. It consists of pointers

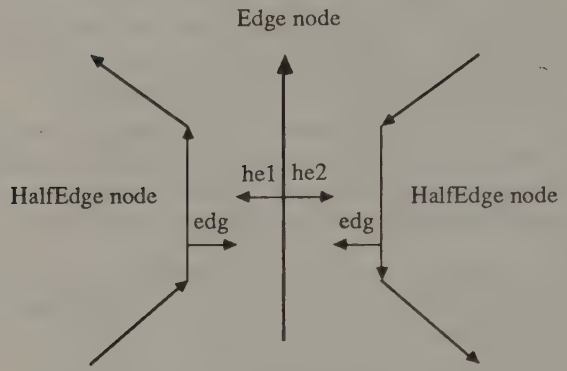



**Figure 10.3** Identification of halfedges.

to the "left" and to the "right" halfedge. The doubly-linked list of edges is realized by means of pointers to the next and the previous edge.

**HalfEdge** Each halfedge includes an additional pointer to its parent edge.

**Vertex** Each vertex includes an additional pointer to one the halfedges emanating from it. To represent the identification of vertices, all faces that have a corner at the some particular point must refer (through loops and halfedges) to a single vertex node corresponding with that point.

The separation of the identification information from face boundary information brought about the division of a full winged-edge node in three nodes makes the representation of various special cases most natural; for instance, an edge occurring twice at a face would be represented as two halfedges referring to the same edge.

Observe that edges can be considered to define an orientation to its halves; that is, the "right" half is considered to be positively oriented, while the "left" half is negatively oriented.

Representation of the identification information is visualized in Figure 10.3 which also gives the C identifiers of some of the required pointers. The three nodes halfedge, edge, and vertex form the most central part of the whole data structure, and the reader is strongly urged to study these interconnections to understand how they work together.

### 10.2.3 Empty Loops

Following the conventions of Chapter 9, we shall allow the special case of an *empty* loop with just one vertex but no edges at all. In the half-edge data structure, empty loops will be represented in terms of one halfedge whose vertex pointer points at the single vertex, and whose edge pointer is NIL. The pointers to the next and the previous halfedge refer to this halfedge itself. Such exceptional halfedges do not correspond with line segments. Observe that empty loops cannot be naturally represented in the hierarchy of Figure 10.2.

## 10.3 IMPLEMENTATION DETAILS

The full C definition of the half-edge data structure is given as the include file `gwb.h` of Program 10.1. All procedures of GWB to be presented in Part Two must have a `"#include "gwb.h""` directive that makes these definitions available, although we shall not show it explicitly in displayed algorithms.

In the definitions, the `typedef` mechanism of C is used to define the nodes of the data structure as new types. This not only leads to cleaner code, but also makes more thorough compilation time type checking possible and enhances portability. The file also defines some general-purpose types such as the type *Id* for vertex and face identifiers and the types *vector* and *matrix* for 4-vectors and 4-by-4 matrices. These types coincide with those used in the vector and matrix package of Appendix B.

The definitions themselves should be self-evident. The "generic" type *Node* is needed in the storage allocator and other low-level routines to be explained in the next chapter.

The file `params.h` referred to in Program 10.1 includes a collection of literals, macros, and global variables. Their use will be described in the following sections. For reference, the file is given in Program 10.2.

## 10.4 ACCESSING THE DATA STRUCTURE

The half-edge data structure is a relatively complicated object, and programs working on it tend to become quite rich of the C "arrowhead" (->) notation. This section seeks to remedy this complexity by discussing the navigation in the data structure.

```
typedef float                   vector[4];
typedef float                   matrix[4][4];
typedef short                   Id;
typedef struct solid            Solid;
typedef struct face             Face;
typedef struct loop             Loop;
typedef struct halfedge         HalfEdge;
typedef struct vertex           Vertex;
typedef struct edge             Edge;
typedef union nodes             Node;

struct solid
{
        Id          solidno;    /* solid identifier */
        Face        *sfaces;    /* pointer to list of faces */
        Edge        *sedges;    /* pointer to list of edges */
        Vertex      *sverts;    /* pointer to list of vertices */
        Solid       *nexts;     /* pointer to next solid */
        Solid       *prevs;     /* pointer to previous solid */
};
struct face
{
        Id          faceno;     /* face identifier */
        Solid       *fsolid;    /* back pointer to solid */
        Loop        *flout;     /* pointer to outer loop */
        Loop        *floops;    /* pointer to list of loops */
        vector      feq;        /* face equation */
        Face        *nextf;     /* pointer to next face */
        Face        *prevf;     /* pointer to previous face */
};
struct loop
{
        HalfEdge    *ledg;      /* ptr to ring of halfedges */
        Face        *lface;     /* back pointer to face */
        Loop        *nextl;     /* pointer to next loop */
        Loop        *prevl;     /* pointer to previous loop */
};
```

**Program 10.1** C definition of the half-edge data structure.

```
struct edge
{
        HalfEdge        *he1;       /* pointer to right halfedge */
        HalfEdge        *he2;       /* pointer to left halfedge */
        Edge            *nexte;     /* pointer to next edge */
        Edge            *preve;     /* pointer to previous edge */
};
struct halfedge
{
        Edge            *edg;       /* pointer to parent edge */
        Vertex          *vtx;       /* pointer to starting vertex */
        Loop            *wloop;     /* back pointer to loop */
        HalfEdge        *nxt;       /* pointer to next halfedge */
        HalfEdge        *prv;       /* pointer to previous halfedge */
};
struct vertex
{
        Id              vertexno;   /* vertex identifier */
        HalfEdge        *vedge;     /* pointer to a halfedge */
        vector          vcoord;     /* vertex coordinates */
        Vertex          *nextv;     /* pointer to next vertex */
        Vertex          *prevv;     /* pointer to previous vertex */
};
union nodes
{
        Solid           s;
        Face            f;
        Loop            l;
        HalfEdge        h;
        Vertex          v;
        Edge            e;
};
# include "params.h"
```

**Program 10.1** C definition of the half-edge data structure (cont.).

```
/* return values and misc constants */
# define        ERROR           -1
# define        SUCCESS         -2
# define        NIL             0
# define        PI              3.141592653589793

/* node type parameters for memory allocation routines */
# define        SOLID           0
# define        FACE            1
# define        LOOP            2
# define        HALFEDGE        3
# define        EDGE            4
# define        VERTEX          5

/* coordinate plane names */
# define        X               0
# define        Y               1
# define        Z               2

/* orientations */
# define        PLUS            0
# define        MINUS           1

/* macros */
# define        mate(he)        (((he) == (he)->edg->he1) ? \
                                (he)->edg->he2 : (he)->edg->he1)
# define        max(x,y)        (((x) > (y)) ? (x) : (y))
# define        abs(x)          (((x) > 0.0) ? (x) : -(x))

/* global variables */
Solid           *firsts;        /* head of the list of all solids */
extern Id       maxs;           /* largest solid no. */
extern Id       maxf;           /* largest face no. */
extern Id       maxv;           /* largest vertex no. */
extern double   EPS;            /* tolerance for geometric tests */
extern double   BIGEPS;         /* a more permissive tolerance */
```

**Program 10.2** Other definitions.

### 10.4.1 Hierarchic Access

Many procedures need simply to scan the whole data structure, and to perform some computation at each node. This is readily possible by following the hierarchy structure of Figure 10.2. For instance, Program 10.3 gives a routine that scans all faces, loops, and halfedges of solid *s*, and lists the vertices of each face.

Through the hierarchy, each vertex could be listed just once by examining its halfedge pointer: if it is NIL, or equals the father halfedge, the vertex is listed; otherwise it is ignored. Similarly, each edge can be listed just once by examining its halfedge pointers. As an alternative of accessing vertices along these lines, the half-edge data structure provides a direct access path to vertices and edges.

### 10.4.2 Access over Edges

The hierarchic access to the half-edge data structure is sufficient for many algorithms—say, generation of graphics output and coordinate transformations. For many algorithms to be discussed in the subsequent chapters this is not sufficient, however.

Edge nodes of the data structure hold the key for more involved navigation in the data structure. In particular, from a halfedge that does not represent an empty loop, they allow the halfedge identified with it to be accessed. This task is so frequent that Program 10.2 defines a macro for it:

```
# define mate(he) (((he) == (he)->edg->he1) ?
                   (he)->edg->he2 : (he)->edg->he1)
```

In other words, if halfedge *he* is the "left" half of its edge, the "mate" halfedge is the right half, and vice versa.

This simple macro allows many apparently nontrivial procedures to be written in a straightforward manner. For instance, the procedure of Program 10.4 accesses all neighbors of a given vertex. Observe how scanning halfedges of a loop (Program 10.3) and scanning halfedges of a vertex become structurally similar to each other. As typical, "lone" vertices (belonging to an empty loop) must be handled as an exception.

## 10.5 NOTES

As presented, the half-edge data structure includes somewhat more data than the bare minimum. If the storage space is very scarce, it may become necessary to remove some of the redundant (but convenient) data.

```
void    listsolid(s)
Solid   *s;
{
     Face          *f;
     Loop          *l;
     HalfEdge      *he;

     f = s->sfaces;
     while(f)
     {
          printf("face %d:\n", f->faceno);
          l = f->floops;
          while(l)
          {
               printf("loop:");
               he = l->ledg;
               do
               {
                    printf(" %d: (%f %f %f)",
                           he->vtx->vertexno,
                           he->vtx->vcoord[0],
                           he->vtx->vcoord[1],
                           he->vtx->vcoord[2]);
               }
               while((he = he->nxt) != l->ledg);
               printf("\n");
               l = l->nextl;
          }
          f = f->nextf;
     }
}
```

**Program 10.3** Hierarchic access.

```
void    listneighbors(v)
Vertex  *v;
{
        HalfEdge        *adj;

        adj = v->vedge;
        if(adj)
                do
                {
                        printf("%d ", adj->nxt->vtx->vertexno);
                }
                while((adj = mate(adj)->nxt) != v->vedge);
        else    printf("no adjacent vertices");
        printf("\n");
}
```

**Program 10.4** Neighbor halfedge access.

The two-way linked lists of faces, edges, and vertices of a solid, and loops of a face can be changed to single-way lists with only minor modifications in the algorithms of GWB and with only small penalties in speed. As explained at the end of Section 10.4.1, the lists of edges and vertices could also be completely dropped with just a little additional overhead.

The next possible change involves the removal of back pointers from half-edges. However, this change would cause substantial revisions in the implementation of Euler operators and many of the procedures directly based on them. As a final revision, nodes *HalfEdge* and *Edge* could be merged into a single node that represents a full winged edge. Again, substantial changes would be required in the algorithms.

## PROBLEMS

10.1. Learn C.

10.2. Modify the data structure definitions of Program 10.1 so that the nodes *HalfEdge* and *Edge* have been merged.

10.3. Rewrite Program 10.3 so as to use the modified data structure of Problem 10.2.

10.4. Rewrite Program 10.4 so as to use the modified data structure of Problem 10.2.

10.5. The inner loops of Programs 10.3 and 10.4 are duals of each other. How well is the dualism preserved in the modified programs of Problems 10.3 amd 10.4?

## BIBLIOGRAPHIC NOTES

The data structure chosen for GWB is by no means the only possibility, and other data structures have been used to support Euler-like operations on boundary models.

For instance, Ansaldi *et al.* [4] describe a data structure called the *Face Adjacency Graph* (FAG) for representing a boundary model. The FAG treats faces as the basic nodes of the data structure, and (as the name suggests) concentrates on representing their adjacency relationships.

# Chapter 11

# IMPLEMENTATION OF EULER OPERATORS

This chapter provides information on the implementation of Euler operators. It describes several layers of procedures needed for this purpose, ranging from level of storage allocation for the data structures to Euler operators themselves.

## 11.1 OVERVIEW

The actual implementation of Euler operators is not a difficult task. Nevertheless, considerable care must be used to make sure that all cases are covered (such as empty loops and "circular" edges). The algorithms of GWB will not at all be afraid of creating the most complicated of cases.

We shall follow a "bottom-up" approach in describing the implementation. Hence we start from the lowest levels of interest for the purposes of this book, and build additional levels on top of them. This leads to a very economical design in terms of the code size (if not necessarily speed).

## 11.2 A STORAGE ALLOCATOR

The most primitive functionality needed for Euler operators consists of storage allocation and deallocation. We shall here build our own allocator on the top of the ordinary UNIX utilities. In practice, this allocator should be replaced by a more rigorous version that itself keeps track of the available storage and avoids the costly UNIX memory allocation procedure `malloc`.



ENGINEERING AND SCIENCE LIBRARY,
P.C.L., 115 NEW CAVENDISH ST., LONDON, W1M 8JS

```
unsigned nodesize[] =
{
        sizeof(Solid), sizeof(Face), sizeof(Loop),
        sizeof(HalfEdge), sizeof(Edge), sizeof(Vertex), 0,
};

Node    *new(what, where)
int     what;
Node    *where;
{
        Node    *node;
        char    *malloc();

        node = (Node *) malloc(nodesize[what]);
        switch(what)
        {
                case SOLID:
                        addlist(SOLID, node, NIL);
                        node->s.sfaces = (Face *) NIL;
                        node->s.sedges = (Edge *) NIL;
                        node->s.sverts = (Vertex *) NIL;
                        break;
                case FACE:
                        addlist(FACE, node, where);
                        node->f.floops = (Loop *) NIL;
                        node->f.flout = (Loop *) NIL;
                        break;
                case LOOP:
                        addlist(LOOP, node, where);
                        break;
                case EDGE:
                        addlist(EDGE, node, where);
                        break;
                case VERTEX:
                        addlist(VERTEX, node, where);
                        node->v.vedge = (Vertex *) NIL;
                        break;
                default:
                        break;
        }
        return(node);
}
```

**Program 11.1** Storage allocation.

A design goal of the allocator is to simplify the implementation of Euler operators by pushing the handling of many of the pointers of the half-edge data structure into the allocator and other very low-level routines. This is accomplished by giving the allocator information on where the new node is to be inserted.

The allocator new of Program 11.1 returns a pointer to a new node of the data structure; good C programming style requires that a type conversion is specified when calling the allocator. The type of the node is selected with argument *what*, which must be one of the literals SOLID, FACE, LOOP, HALFEDGE, EDGE, or VERTEX. Another argument *where* must point to a parent node of the new node as follows:

1. *what* = FACE, EDGE, or VERTEX: *where* points to the parent solid.

2. *what* = LOOP: *where* points to the parent face.

In these cases new performs the necessary manipulations to link the new node with its parent. Otherwise *where* has no effect and may be NIL. Array *nodesize* contains the sizes of the various nodes.

Analogously to the storage allocator new, the deallocator procedure del(what, which, where) removes the node *which* from the data structure, and disconnects it from the list of the children of the parent node *where* if *what* = SOLID, FACE, EDGE, LOOP, or VERTEX. All our algorithms will assume that the data in a deleted node remains valid until the next call of new. We leave the implementation of del to the reader.

### 11.2.1 Linked List Procedures

The actual insertion of faces, loops, edges, and vertices into the proper lists takes place with the procedure addlist given in Program 11.2. It adds the node pointed to by *which* into the linked list of nodes owned by *where*. As a special case, solids are simply linked to the list of all solids; in this case, *where* has no effect. The type of *which* is again given in terms of *what* and must be one of the literals SOLID, FACE, LOOP, EDGE, or VERTEX.

To unlink children from their parents, the corresponding inverse procedure dellist(what, which, where) is used. Specifically, the node *which* of type *what* (SOLID, FACE, LOOP, EDGE, VERTEX) will be removed from the respective linked list of the parent node *which*. Again, solids are an exception. Obviously, dellist would be used in the implementation of del in an analogous way to addlist and new. The implementation of dellist is not elaborated here.

```
void  addlist(what, which, where)
int   what;
Node  *which, *where;
{
    switch(what)
    {
        case SOLID:
            which->s.nexts = firsts;
            which->s.prevs = (Solid *) NIL;
            if(firsts)
                firsts->prevs = (Solid *) which;
            firsts = (Solid *) which;
            break;
        case FACE:
            which->f.nextf = where->s.sfaces;
            which->f.prevf = (Face *) NIL;
            if(where->s.sfaces)
                where->s.sfaces->prevf = (Face *) which;
            where->s.sfaces = (Face *) which;
            which->f.fsolid = (Solid *) where;
            break;
        case LOOP:
            which->l.nextl = where->f.floops;
            which->l.prevl = (Loop *) NIL;
            if(where->f.floops)
                where->f.floops->prevl = (Loop *) which;
            where->f.floops = (Loop *) which;
            which->l.lface = (Face *) where;
            break;
        case EDGE:
            which->e.nexte = where->s.sedges;
            which->e.preve = (Edge *) NIL;
            if(where->s.sedges)
                where->s.sedges->preve = (Edge *) which;
            where->s.sedges = (Edge *) which;
            break;
        case VERTEX:
            which->v.nextv = where->s.sverts;
            which->v.prevv = (Vertex *) NIL;
            if(where->s.sverts)
                where->s.sverts->prevv = (Vertex *) which;
            where->s.sverts = (Vertex *) which;
            break;
    }
}
```

**Program 11.2** Linked list manipulation.

### 11.2.2 Halfedge Procedures

In the implementation of the allocator, the actual initialization of halfedges and edges is performed by procedure **addhe**. Referring to Program 11.3, the procedure allocates a new halfedge, links it into loop *l* at the front of an existing halfedge *where*, and sets the other pointers appropriately. Note that **addhe** is written so as to handle the special cases of an empty original loop (`where->edg` = `NIL`) or no loop at all (`where` = `NIL`).

The inverse procedure **delhe** of Program 11.4 performs the opposite actions. Note in particular how it creates an "empty" loop in the case that `he->edg` $\neq$ `NIL` and `he->nxt` = `he`.

Algorithms of GWB will do the bulk of their work with Euler operators. Occasionally, however, they will also use some of the functions described above. For instance, the sectioning algorithm of Chapter 14 will use **new**, **addlist**, and **dellist** to actually divide the sectioned solid.

## 11.3 LOW-LEVEL EULER OPERATORS

Euler operators will be available in GWB in two versions. In the lower layer, the operators are specified by means of pointers to adjacent nodes of the half-edge data structure, and they can work without any searching of the data structure. In the upper layer, the operators will be parameterized in terms of face and vertex identifiers, and they will have to search for the relevant nodes of the data structure.

Based on the storage allocator and related procedures, this section describes the implementation of low-level Euler operators, while the high-level operators will be the topic of the next section. Because of space limitations, it is not possible to give a fully detailed implementation of each operator. Instead, we give three examples that should give a good idea of how the remaining ones should be written.

### 11.3.1 MVFS

Logically the first Euler operator we need is **mvfs**, the initialization Euler operator. Based on the procedures developed in the preceding sections, the concise implementation of Program 11.5 can readily be given.

Referring to the program, the procedure allocates one each of nodes *Solid*, *Face*, *Loop*, and *Vertex* with **new**, and one *HalfEdge* with **addhe**. Note how **new** will take care of linking child nodes properly into their parents. A new halfedge is properly initialized to represent the "empty" loop consisting of one vertex but no edges at all. The reader is encouraged to sketch with pencil and paper the resulting data structure.

```
HalfEdge        *addhe(e, v, where, sign)
Edge            *e;
Vertex          *v;
HalfEdge        *where;
int             sign;
{
        HalfEdge        *he;

        if(where->edg == NIL)
        {
                he = where;
        }
        else
        {
                he = (HalfEdge *) new(HALFEDGE, NIL);
                where->prv->nxt = he;
                he->prv = where->prv;
                where->prv = he;
                he->nxt = where;
        }

        he->edg = e;
        he->vtx = v;
        he->wloop = where->wloop;
        if(sign == PLUS)
                e->he1 = he;
        else    e->he2 = he;

        return(he);
}
```

**Program 11.3** Halfedge manipulation.

```
HalfEdge        *delhe(he)
HalfEdge        *he;
{
        if(he->edg == NIL)
        {
                del(HALFEDGE, he, NIL);
                return(NIL);
        }
        else if(he->nxt == he)
        {
                he->edg = NIL;
                return(he);
        }
        else
        {
                he->prv->nxt = he->nxt;
                he->nxt->prv = he->prv;
                del(HALFEDGE, he, NIL);
                return(he->prv);
        }
}
```

**Program 11.4** Halfedge deletion.

```
Solid   *mvfs(s, f, v, x, y, z)
Id      s, f, v;
float   x, y, z;
{
        Solid           *newsolid;
        Face            *newface;
        Vertex          *newvertex;
        HalfEdge        *newhe;
        Loop            *newloop;

        newsolid = (Solid *) new(SOLID, NIL);
        newface = (Face *) new(FACE, newsolid);
        newloop = (Loop *) new(LOOP, newface);
        newvertex = (Vertex *) new(VERTEX, newsolid);
        newhe = (HalfEdge *) new(HALFEDGE, NIL);

        newsolid->solidno = s;
        newface->faceno = f;
        newface->flout = newloop;
        newloop->ledg = newhe;
        newhe->wloop = newloop;
        newhe->nxt = newhe->prv = newhe;
        newhe->vtx = newvertex;
        newhe->edg = NIL;
        newvertex->vertexno = v;
        newvertex->vcoord[0] = x;
        newvertex->vcoord[1] = y;
        newvertex->vcoord[2] = z;
        newvertex->vcoord[3] = 1.0;

        return(newsolid);
}
```

**Program 11.5** The mvfs operator.

### 11.3.2 LMEV

As the next example, we pick the operator `lmev`, the low-level vertex-splitting operator. Recall that we expect `lmev` to be capable of dealing with all special cases of empty loops, "strut" edges, and so on. It turns out that this can be implemented in a relatively natural and concise fashion.

Referring to the code of Program 11.6, if the arguments *he1* and *he2* of the procedure differ, `lmev` will divide the cycle of halfedges of vertex `he1->vtx` in two cycles so that *he1* is the first halfedge in one cycle, and *he2* in the other. The identifier *vn*, the coordinates *x y z* and the cycle starting at *he1* are assigned to a new vertex. A new edge joins `he1->vtx` and the new vertex. In the case that *he1* = *he2*, a "strut" edge that appears twice at a loop is added at front of *he1*.

Nicely enough, the halfedge manipulation procedures allow this specification to be implemented without any special code for different cases. The procedure first allocates and initializes the new vertex and edge nodes. As halfedges from *he1* (inclusive) to *he2* (noninclusive) will be assigned to a new vertex, the `while`-loop updates their vertex pointers. Note how macro `mate` is again used to select the next halfedge of the cycle. Finally, new halfedges are inserted, and the cycle pointers of vertices updated. Note that the new edge will be oriented towards the old vertex. Again, the reader is encouraged to check the effects of the code in various cases (e.g., when applied to the halfedge created by `mvfs`).

### 11.3.3 LMEF

The next example is the code for `lmef`, the low-level face splitting operator. Dually to `lmev`, if *he1* $\neq$ *he2*, `lmef` is expected to divide the loop of *he1* and *he2* into two loops with a new edge from `he1->vtx` to `he2->vtx`. Halfedge *he2* remains in the "old" loop, while *he1* is moved to the new loop that becomes the outer boundary of a new face. In the special case that *he1* = *he2*, a "circular" face with just one edge is created.

The code implementing this is given in Program 11.7. It is written so as to stress the dualism of MEF and MEV, and the reader is urged to compare Programs 11.6 and 11.7 closely. This code splits the original loop by first inserting the two halfedges as usually, and then "swapping" their tails. This has the nice feature that no special code for "circular" new faces (i.e., case *he1* = *he2*) is needed.

```
void            lmev(he1, he2, v, x, y, z)
HalfEdge        *he1, *he2;
Id              v;
float           x, y, z;
{
        HalfEdge        *he;
        Vertex          *newvertex;
        Edge            *newedge;

        newedge = (Edge *) new(EDGE, he1->wloop->lface->fsolid);
        newvertex = (Vertex *) new(VERTEX,
                he1->wloop->lface->fsolid);
        newvertex->vcoord[0] = x;
        newvertex->vcoord[1] = y;
        newvertex->vcoord[2] = z;
        newvertex->vcoord[3] = 1.0;
        newvertex->vertexno = v;

        he = he1;
        while(he != he2)
        {
                he->vtx = newvertex;
                he = mate(he)->nxt;
        }

        addhe(newedge, newvertex, he2, PLUS);
        addhe(newedge, he2->vtx, he1, MINUS);

        newvertex->vedge = he2->prv;
        he2->vtx->vedge = he2;
}
```

**Program 11.6** The lmev operator.

```
Face            *lmef(he1, he2, f)
HalfEdge        *he1, *he2;
Id              f;
{
        Face            *newface;
        Loop            *newloop;
        Edge            *newedge;
        HalfEdge        *he, *nhe1, *nhe2, *temp;

        newface = (Face *)new(FACE, he1->wloop->lface->fsolid);
        newloop = (Loop *)new(LOOP, newface);
        newedge = (Edge *)new(EDGE, he1->wloop->lface->fsolid);
        newface->faceno = f;
        newface->flout = newloop;

        he = he1;
        while(he != he2)
        {
                he->wloop = newloop;
                he = he->nxt;
        }

        nhe1 = addhe(newedge, he2->vtx, he1, MINUS);
        nhe2 = addhe(newedge, he1->vtx, he2, PLUS);

        nhe1->prv->nxt = nhe2;
        nhe2->prv->nxt = nhe1;
        temp = nhe1->prv;
        nhe1->prv = nhe2->prv;
        nhe2->prv = temp;

        newloop->ledg = nhe1;
        he2->wloop->ledg = nhe2;

        return(newface);
}
```

**Program 11.7** The lmef operator.

### 11.3.4 LKEMR

The final example is the code of `lkemr`, the low-level operator for splitting a loop into two components.

The actual splitting of the loop is implemented in Program 11.8 in the most straightforward fashion. After a new loop node has been allocated, the two halfedges *h1* and *h2* corresponding with the edge to be removed are manipulated so as to effectively leave them in two distinct components, of which the one associated with *h2* becomes the new loop. This manipulation has the virtue that `delhe` can be used to actually delete *h1* and *h2*, and the various special cases of empty loops are eliminated.

Operators that remove edges (`lkev`, `lkef`, `lkemr`) must be particularly carefully written. In this case, it might happen that the loop nodes refer to a halfedge just deleted, so the *ledg*-pointers of the loops must be updated. Similarly, the vertices of *h1* and *h2* may refer back to *h1* and *h2* through the *vedge*-pointer.

## 11.4 HIGH-LEVEL EULER OPERATORS

The implementation of high-level Euler operators is based on their low-level counterparts and auxiliary procedures that search the half-edge data structure in order to find all the required pointers. In the simple implementation presented below, all searches are implemented as sequential scans. When dealing with solids with more than, say, 100 faces, an implementation allowing direct access to faces based on hashing should be used.

Referring to Program 11.9, the procedure `getsolid` will locate the solid with the identifier *sn* from the list of all solids. If the procedure is successful, it returns a pointer to the solid; otherwise, an empty pointer is returned.

Similarly, the procedure `fface` locates the face of *s* with identifier *fn*, and returns a pointer to it. After getting our hands on a face *f*, procedure `fhe` seeks a halfedge from *vn1* to *vn2*. Again, if these procedures cannot find what they seek, they return `NIL`.

With the scanning procedures, the implementation of high-level Euler operators is straightforward. For instance, the code for the high-level operator `mev` is given in Program 11.10. After finding the desired pointers, the procedure simply calls the corresponding low-level operator `lmev` to do the job. Most of the code is actually spent in catching errors in solid, face, or vertex identifiers. The procedure returns a code indicating the success or failure of the operation requested. It is good coding practice to give all high-level Euler operators this capability.

```
void            lkemr(h1, h2)
HalfEdge        *h1, *h2;
{
        register HalfEdge       *h3, *h4;
        Loop                    *nl;
        Loop                    *ol;
        Edge                    *killedge;

        ol = h1->wloop;
        nl = (Loop *) new(LOOP, ol->lface);
        killedge = h1->edg;

        h3 = h1->nxt;
        h1->nxt = h2->nxt;
        h2->nxt->prv = h1;
        h2->nxt = h3;
        h3->prv = h2;

        h4 = h2;
        do
        {
                h4->wloop = nl;
        }
        while((h4 = h4->nxt) != h2);

        ol->ledg = h3 = delhe(h1);
        nl->ledg = h4 = delhe(h2);

        h3->vtx->vedge = (h3->edg) ? h3 : (HalfEdge *) NIL;
        h4->vtx->vedge = (h4->edg) ? h4 : (HalfEdge *) NIL;

        del(EDGE, killedge, ol->lface->fsolid);
}
```

**Program 11.8** The lkemr operator.

```
Solid     *getsolid(sn)
Id        sn;
{
          Solid   *s;
          for(s = firsts; s != NIL; s = s->nexts)
                  if(s->solidno == sn)
                          return(s);
          return(NIL);
}

Face      *fface(s, fn)
Solid     *s;
Id        fn;
{
          Face    *f;
          for(f = s->sfaces; f != NIL; f = f->nextf)
                  if(f->faceno == fn)
                          return(f);
          return(NIL);
}

HalfEdge  *fhe(f, vn1, vn2)
Face      *f;
Id        vn1, vn2;
{
          Loop            *l;
          HalfEdge        *h;
          for(l = f->floops; l != NIL; l = l->nextl)
          {
                  h = l->ledg;
                  do
                  {
                          if(h->vtx->vertexno == vn1 &&
                             h->nxt->vtx->vertexno == vn2)
                                  return(h);
                  }
                  while((h = h->nxt) != l->ledg);
          }
          return(NIL);
}
```

**Program 11.9** Scanning procedures.

```
int     mev(s, f1, f2, v1, v2, v3, v4, x, y, z)
Id      s, f1, f2, v1, v2, v3, v4;
float   x, y, z;
{
      Solid         *oldsolid;
      Face          *oldface1, *oldface2;
      HalfEdge      *he1, *he2;

      if((oldsolid = getsolid(s)) == NIL)
      {
            fprintf(stderr, "mev: solid %d not found\n", s);
            return(ERROR);
      }
      if((oldface1 = fface(oldsolid, f1)) == NIL)
      {
            fprintf(stderr,
                  "mev: face %d not found in solid %d\n", f1, s);
            return(ERROR);
      }
      if((oldface2 = fface(oldsolid, f2)) == NIL)
      {
            fprintf(stderr,
                  "mev: face %d not found in solid %d\n", f2, s);
            return(ERROR);
      }
      if((he1 = fhe(oldface1, v1, v2)) == NIL)
      {
            fprintf(stderr,
                  "mev: edge %d-%d not found in face %d\n",
                        v1, v2, f1);
            return(ERROR);
      }
      if((he2 = fhe(oldface2, v1, v3)) == NIL)
      {
            fprintf(stderr,
                  "mev: edge %d-%d not found in face %d\n",
                        v1, v3, f2);
            return(ERROR);
      }
      lmev(he1, he2, v4, x, y, z);
      return(SUCCESS);
}
```

**Program 11.10** The mev operator.

## 11.5 C CALLING SEQUENCES

This section specifies the details of the programming interface to all Euler operators and related procedures as used in Part Two by giving summaries of their calling sequences from C programming language. In some cases the effects are depicted in terms of partial plane models; in all such illustrations, arrowheads are used to denote the orientations of halfedges and edges. Orientations of all faces are clockwise.

### 11.5.1 Low-Level Euler Operators

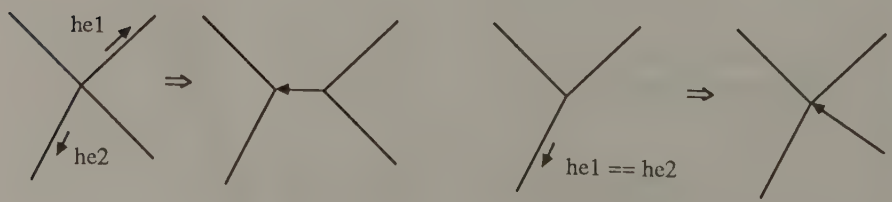
```
Solid    *mvfs(sn, f, v, x, y, z)
Id       sn, f, v;
float    x, y, z;
```

Operator `mvfs` creates a new solid named *sn* that consists of one face *f* and one vertex *v* with coordinates *x*, *y*, and *z*. It returns a pointer to the new solid created. Note that the operator `mvfs` is the only one that has just one level of implementation: there are no separate low-level and high-level operators.

```
void     lkvfs(s)
Solid    *s;
```

Operator `lkvfs` is the inverse of `mvfs`, and removes the solid pointed at by *s*. The solid is assumed to consist of a single face and vertex only.

```
void     lmev(he1, he2, v, x, y, z)
HalfEdge *he1, *he2;
Id       v;
float    x, y, z;
```

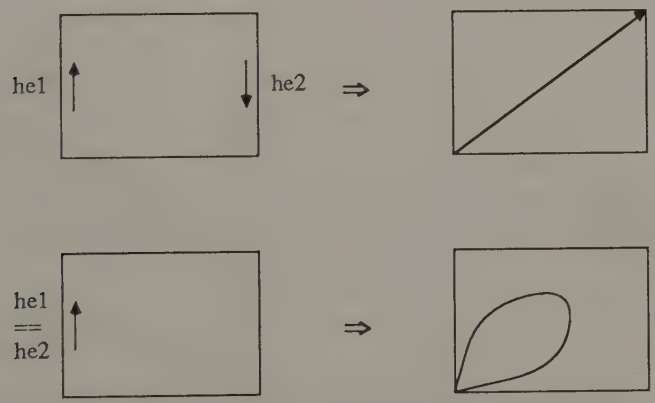


Operator `lmev` "splits" the vertex pointed at by *he1* and *he2*, and adds a new edge between the resulting two vertices. The identifier *v* and the coordinates *x*, *y*, and *z* are assigned to the new vertex. The case that *he1* and *he2* are identical is allowed; in this case, the new vertex and edge are added into the face of *he1*. The new edge is oriented from the new vertex to the old one.

```
void     lkev(he1, he2)
HalfEdge *he1, *he2;
```

Operator `lkev` is the inverse of `lmev`. It removes the edge pointed at by both *he1* and *he2*, and "joins" the two vertices `he1->vtx` and `he2->vtx` which must be distinct (but can, of course, correspond with geometrically identical points). Vertex `he1->vtx` is removed.

```
Face     *lmef(he1, he2, f)
HalfEdge *he1, *he2;
Id       f;
```




Operator `lmef` adds a new edge between halfedges *he1* and *he2*, and "splits" their common face into two faces such that *he1* will occur in the new face *f*, and *he2* remains in the old face. The new edge is oriented from `he1->vtx` to `he2->vtx`. Halfedges *he1* and *he2* must belong to the same loop (i.e., `he1->wloop == he2->wloop`). They may be equal, in which case a "circular"

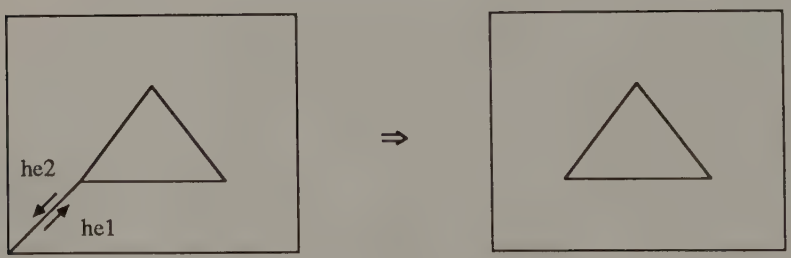
face with just one edge is created. A pointer to the new face is returned.

Observe that `lmef` does not attempt to classify the loops of the divided face into the faces resulting from the division, i.e., all other loops of the old face remain in it. The auxiliary procedure `ringmv` described at the end of this section can be used to move the loops to the new face, if desired.

```
void     lkef(he1, he2)
HalfEdge *he1, *he2;
```

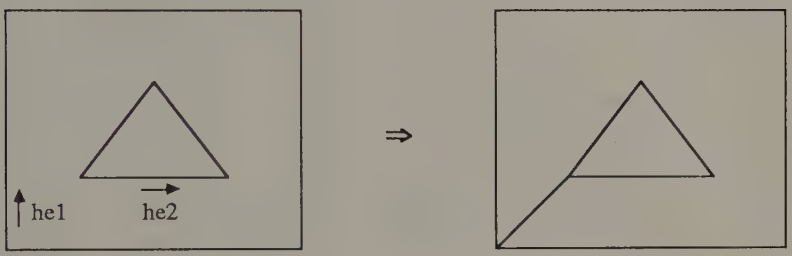
Operator `lkef` is the inverse of `lmef`. It removes the edge of *he1* and *he2*, and "joins" the two adjacent faces by merging their loops. The face `he2->wloop->lface` is removed. `lkef` is applicable to the halves of an edge that occurs in two distinct faces. That is, it is assumed that `he1->edg == he2->edg` and `he1->wloop->lface != he2->wloop->lface`.

```
void     lkemr(he1, he2)
HalfEdge *he1, *he2;
```




Operator `lkemr` removes the edge of *he1* and *he2*, and divides their common loop into two components (i.e., it creates a new "ring"). If the original loop was "outer," the component of `he1->vtx` becomes the new "outer" loop. (If this default is inappropriate, you can simply swap the arguments of `lkemr`, or use `lringmv` to make the desired loop "outer".) It is assumed that `he1->edg == he2->edg` and `he1->wloop == he2->wloop`.

```
void      lmekr(he1, he2)
HalfEdge *he1, *he2;
```




Operator `lmekr` is the inverse of `lkemr`. It inserts a new edge between the vertices of *he1* and *he2*, and merges the corresponding loops into one loop (i.e., removes a ring). The operator assumes that `he1->wloop != he1->wloop`, i.e., *he1* and *he2* belong to two distinct loops, and that `he1->wloop->lface == he2->wloop->lface`, i.e., they occur in a single face.

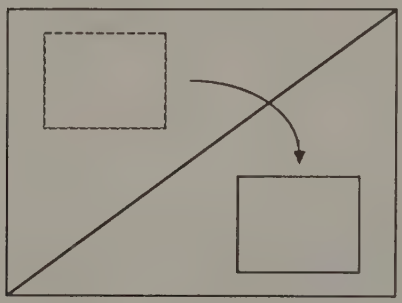
```
void      lkfmrh(fac1, fac2)
Face      *fac1, *fac2;
```

Operator `lkfmrh` merges two faces *fac1* and *fac2* by making the loop of the latter a ring into the former. Face *fac2* is hence removed. It is assumed that *fac2* is simple, i.e., has just one loop.

```
void      lmfkrh(l, f)
Loop      *l;
Id        f;
```

Operator `lmfkrh` is the inverse of `lkfmrh`. It makes the loop *l* the outer loop of a new face *f*. It is assumed that *l* is an inner loop of its parent face.

```
void    lringmv(l, tofac, inout)
Loop    *l;
Face    *tofac;
int     inout;
```



Procedure `lringmv` moves the loop *l* from its parent face to face *tofac*. If *inout* = 0 *l* becomes an "inner" loop, and otherwise the outer loop of *tofac*. If `l->lface == tofac`, `lringmv` has no effect except for assigning the inner vs. outer information. Note that `lringmv` is an addendum to `lmef`, and not an Euler operator.

### 11.5.2 High-Level Euler Operators

All high-level operators take a list of vertex, face, and solid identifiers as their input. Having no names themselves, edges and halfedges are identified through their vertices. The notation *v1* → *v2* is used below to denote the edge between vertices *v1* and *v2*.

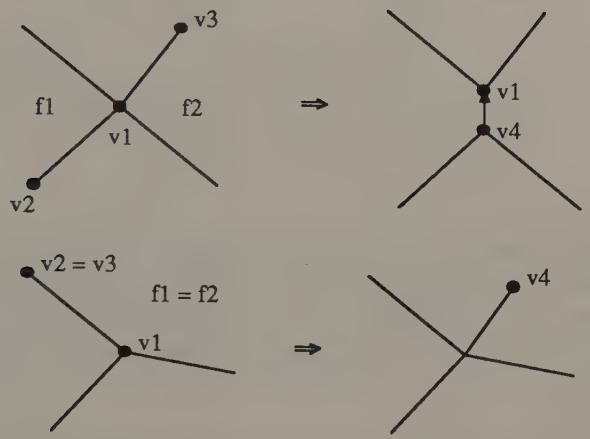
The parameterization of high-level operators can be quite mechanically derived from the low-level ones by replacing pointers to halfedges by a triple (*f*, *v1*, *v2*), where *v1* → *v2* is an edge appearing in the face *f*. Because this leads to quite complicated parameter lists, alternative parameterizations for some operators will be given.

```
void    kvfs(s)
Id      s;
```

Operator `kvfs` is the inverse of `mvfs`, and removes the solid *s*.

```
int     mev(s, f1, f2, v1, v2, v3, v4, x, y, z)
Id      s, f1, f2, v1, v2, v3, v4;
float   x, y, z;

int     smev(s, f1, v1, v4, x, y, z)
Id      s, f1, v1, v4;
float   x, y, z;
```




Operator mev divides the cycle of edges around the vertex *v1* so that all edges from *v1* → *v2* (inclusive) to *v1* → *v3* (exclusive) will become adjacent to a new vertex *v4*. The vertices *v1* and *v4* are joined with a new edge. Coordinates *x*, *y*, and *z* are assigned to *v4*.

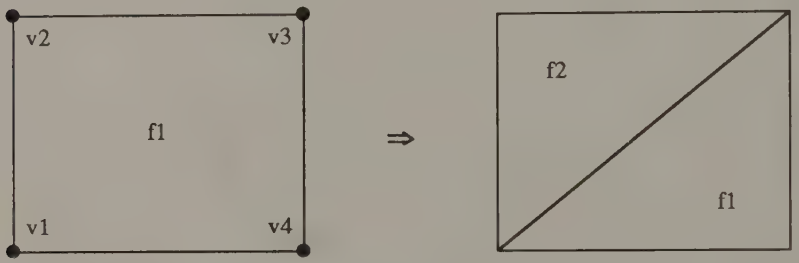
Similarly to the corresponding low-level operator, various special cases are possible. Of particular use is the "line-drawing" case that *f1* = *f2* and *v2* = *v3*. In this case, a new edge to a new vertex will be added within the face. We shall call such "dangling" edges *struts*. This case occurs so frequently that we include a "convenience" procedure smev that performs this operation. The procedure assumes that *v1* appears just once in *f1*; hence, the argument *v2* can be left out.

```
int     kev(s, f, v1, v2)
Id      s, f, v1, v2;
```

Operator kev is the exact inverse of mev. It removes the edge *v1* → *v2* from face *f* by merging *v2* with *v1*.

```
int     mef(s, f1, v1, v2, v3, v4, f2)
Id      s, f1, f2, v1, v2, v3, v4;

int     smef(s, f1, v1, v3, f2)
Id      s, f1, f2, v1, v3;
```

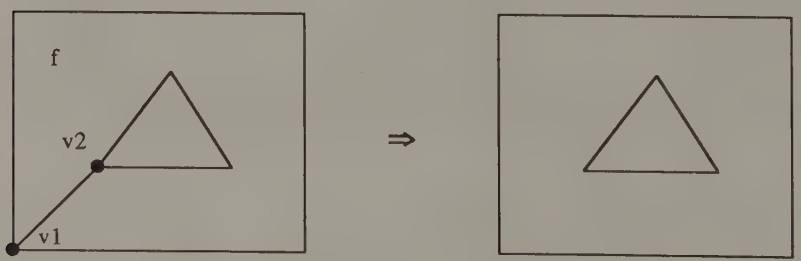



Operator **mef** connects the vertices *v1* and *v3* of face *f1* with a new edge, and creates a new face *f2*. Similarly to **smev**, we include the convenience procedure **smef** that leaves the arguments *v1* and *v4* out. It should be applied only if *v1* and *v3* are known to occur just once in the face.

```
int     kef(s, f, v1, v2)
Id      s, f, v1, v2;
```

Operator **kef** removes the edge $v1 \rightarrow v2$, and joins the two faces separated by it.

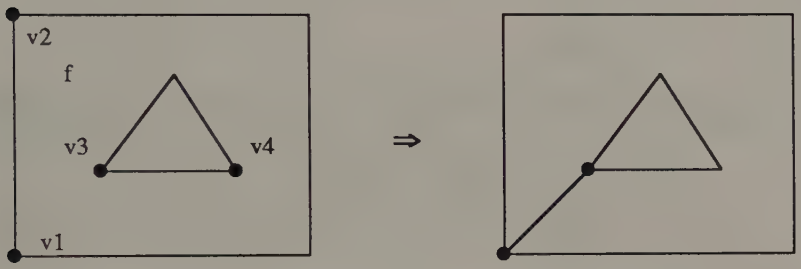
```
int     kemr(s, f, v1, v2)
Id      s, f, v1, v2;
```




Operator **kemr** removes the edge $v1 \rightarrow v2$, and divides the loop that contained it into two parts.

```
int     mekr(s, f, v1, v2, v3, v4)
Id      s, f, v1, v2, v3, v4;

int     smekr(s, f, v1, v3)
Id      s, f, v1, v3;
```




Operator mekr, the inverse of kemr, joins two vertices *v1* and *v3* from distinct loops of face *f* with an edge. The convenience procedure can be used if *v1* and *v3* are known to occur just once in their loops.

```
int     kfmrh(s, f1, f2)
Id      s, f1, f2;
```

Operator kfmrh "merges" two faces *f1* and *f2* by making the latter an interior loop of the former. Face *f2* is removed.

```
int     mfkrh(s, f1, vi, v2, f2)
Id      s, f1, vi, v2, f2;
```

Operator mfkrh makes the interior loop of face *f1* which contains the edge $v1 \rightarrow v2$ the boundary of a new face *f2*. It is the inverse of kfmrh.

```
int     ringmv(s, f1, f2, v1, v2, inout)
Solid   *s;
Id      f1, f2. v1, v2, inout;
```

The procedure ringmv moves a ring from face *f1* to face *f2*. The ring moved is the one that has the edge $v1 \rightarrow v2$. An empty loop is denoted by $v2 = 0$. If $inout = 1$ the moved loop becomes the "outer" loop of *f2*. The procedure is the high-level counterpart of lringmv.

## PROBLEMS

11.1. Implement the list removal procedure **dellist** that has the following C calling sequence:

```
void   dellist(what, which, where)
int    what;
Node   *which;
Node   *where;
```

11.2. Implement the storage deallocator **del** that has the following C calling sequence:

```
void   del(what, which, where)
int    what;
Node   *which;
Node   *where;
```

(You will need the list deletion procedure **dellist** of Problem 11.1.)

11.3. Implement the operators **lkev** and **lkef** obeying the calling sequences of Section 11.5.2.

11.4. Implement the operator **lmekr** according to its functional specification in Section 11.5.2.

11.5. Implement the operators **lkfmrh** and **lmfkrh** as defined in Section 11.5.2.

# Chapter 12

# BASIC MODELING ALGORITHMS

The half-edge data structure and the Euler operators form the basic building blocks of our modeling system. In this chapter we shall start to investigate various higher-level procedures that can be built on the top of them.

## 12.1 MOTIVATION

According to the analogy suggested in Section 8.2.2, Euler operators and their support routines take the role of an "assembly language" in the architecture of GWB. Just as we do not (usually) want to use an assembly-level tool for expressing complicated processing, we shall need to develop various higher levels of modeling tools for geometric modeling.

One of the major virtues of boundary representations is the opportunity to use various sorts of "local modifications" that can generate frequently needed solids without the large computational burden of, say, Boolean set operations. This chapter takes a brief look into a family of such "medium-level" operations useful for implementing parameterized primitives and some drawing-type operations directly on top of Euler operators.

As usual, we aim at a concise and economical implementation. As will be seen, this leads to a situation where the various procedures rely on each other in a fashion that makes it appropriate to consider these routines an independent layer of its own right in the architecture of GWB.

## 12.2 ARC GENERATOR

A typical case for a higher-level operation that should be implemented on the top of Euler operators is the generation a polygonal approximation of a circular arc. Obviously, all arc generation methods discussed in Chapter 2 are of potential interest, and a serious modeler would implement most of them and probably others.

Here we shall discuss only the simplest case, and develop a procedure that can generate an arc based on the radius and the coordinates of the center. Even this is subject to the restriction that the plane of definition is of the form $z = h$. Furthermore, we assume that the first vertex of the arc already exists, and supply its identifier to the procedure.

This specification is implemented in Program 12.1. The procedure **arc** generates an approximation of a circular arc segment with $n$ edges, centered at $(cx\ cy)$, on plane $z = h$, and with radius *rad*. The arc ranges from angle *phi1* to *phi2*, measured in degrees, where 0.0 = $x$-axis and angles grow counterclockwise. The arc starts from existing vertex $v$ of face $f$.

The algorithm must be capable of producing unique new vertex names. Simplistically, this is implemented by means of a procedure **getmaxnames** that stores the largest vertex and face identifiers of its argument solid in global variables *maxv* and *maxf*, respectively.

As given, the procedure is not capable of producing a full circle, or closing a polygon. A general solution must check whether the last point of the arc actually is a vertex of $f$, and if so, apply MEF instead of MEV to generate the last edge of the arc. To just generate a circle, a simpler alternative is to build a circle generator on the top of **arc** as in Program 12.2.

## 12.3 SWEEPING PRIMITIVES

Translational and rotational sweeping are other kinds of convenient solid description techniques that can be implemented in terms of Euler operators.

### 12.3.1 Translational Sweeping

Let us first consider the *face sweep* operation that takes a face $f$ of a solid, and sweeps it along a vector $[dx\ dy\ dz]$. Note that the profile sweep operation is a special case of this, because a closed profile is represented as a *lamina* having two faces like the two sides of a piece of paper. Hence profile sweep amounts to sweeping one or the other of the faces of the lamina.

In this algorithm low-level Euler operators are the most natural tools. The algorithm of Program 12.3 traverses each loop of the argument face,

```
void    arc(s, f, v, cx, cy, rad, h, phi1, phi2, n)
Id      s, f, v;
float   cx, cy, rad, h, phi1, phi2;
int     n;
{
        float   x, y, angle, inc;
        Id      prev;
        int     i;

        angle = phi1 * PI / 180.0;
        inc = (phi2 - phi1) * PI / (180.0 * n);
        prev = v;
        getmaxnames(s);
        for(i = 0; i < n; i++)
        {
                angle += inc;
                x = cx + cos(angle) * rad;
                y = cy + sin(angle) * rad;
                smev(s, f, prev, ++maxv, x, y, h);
                prev = maxv;
        }
}

void    getmaxnames(sn)
Id      sn;
{
        Solid   *s;
        Vertex  *v;
        Face    *f;

        s = getsolid(sn);
        for(v = s->sverts, maxv = 0; v != NIL; v = v->nextv)
                if(v->vertexno > maxv) maxv = v->vertexno;
        for(f = s->sfaces, maxf = 0; f != NIL; f = f->nextf)
                if(f->faceno > maxf) maxf = f->faceno;
}
```

**Program 12.1** Arc generator.

```
Solid   *circle(sn, cx, cy, rad, h, n)
Id      sn;
float   cx, cy, rad, h;
int     n;
{
        Solid   *s;

        s = mvfs(sn, 1, 1, cx+rad, cy, h);
        arc(sn, 1, 1, cx, cy, rad, h, 0.0, ((n-1)*360.0/n), n-1);
        smef(sn, 1, n, 1, 2);
        return(s);
}
```

**Program 12.2** Circle generator.

and adds a new edge from each vertex along the sweep direction with `lmev`, the low-level counterpart of MEV. The end vertices of these new edges are joined with `lmef` to create side faces of the swept region. The algorithm is implemented so that the final top face of the swept region will have the identical identifier as the original face.

As an example of the use of **sweep**, the rectilinear box and cylinder primitives can be implemented as in Program 12.4. Because **sweep** needs a pointer to the argument face, the procedure **fface** described in Chapter 11 is used.

### 12.3.2 Rotational Sweeping

Rotational sweeping "swings" a two-dimensional figure around an axis to create a rotational solid. Of course, we shall create a polyhedral approximation of the rotational surface only; the accuracy of the approximation is selected by parameter *nfac* that tells how many points are used to approximate any circle of the solid.

To keep things simple, let us first consider the most straightforward case where the rotation axis is the $x$-axis, and the figure is a simple, connected, and nonforking string of edges, all lying on the half-plane $y > 0, z = 0$. Such an object is called a *wire*.

The algorithm we shall develop performs the sweeping operation in two phases. Referring to Figure 12.1(a–c), in the first phase the algorithm makes $nfaces - 1$ scans along the wire, and creates one new face for each original wire edge with an algorithm somewhat similar to the translational sweeping algorithm. After the final scan, we have the arrangement of Fig-

```
void    sweep(fac, dx, dy, dz)
Face    *fac;
float   dx, dy, dz;
{
        Loop            *l;
        HalfEdge        *first, *scan;
        Vertex          *v;

        getmaxnames(fac->fsolid);
        l = fac->floops;
        while(l)
        {
                first = l->ledg;
                scan = first->nxt;
                v = scan->vtx;
                lmev(scan, scan, ++maxv,
                        v->vcoord[0] + dx,
                        v->vcoord[1] + dy,
                        v->vcoord[2] + dz);
                while(scan != first)
                {
                        v = scan->nxt->vtx;
                        lmev(scan->nxt, scan->nxt, ++maxv,
                                v->vcoord[0] + dx,
                                v->vcoord[1] + dy,
                                v->vcoord[2] + dz);
                        lmef(scan->prv, scan->nxt->nxt, ++maxf);
                        scan = mate(scan->nxt)->nxt;
                }
                lmef(scan->prv, scan->nxt->nxt, ++maxf);
                l = l->nextl;
        }
}
```

**Program 12.3** Translational sweeping.

```
Solid   *block(sn, dx, dy, dz)
Id      sn;
float   dx, dy, dz;
{
        Solid        *s;

        s = mvfs(sn, 1, 1, 0.0, 0.0, 0.0);
        smev(sm, 1, 1, 2, dx, 0.0, 0.0);
        smev(sn, 1, 2, 3, dx, dy, 0.0);
        smev(sn, 1, 3, 4, 0.0, dy, 0.0);
        smef(sn, 1, 1, 4, 2);
        sweep(fface(s, 2), 0.0, 0.0, dz);
        return(s);
}

Solid   *cyl(sn, rad, h, n)
Id      sn;
float   rad, h;
int     n;
{
        Solid        *s;

        s = circle(sn, 0.0, 0.0, rad, 0.0, n);
        sweep(fface(s, 2), 0.0, 0.0, h);
        return(s);
}
```

**Program 12.4** Block and cylinder primitives.

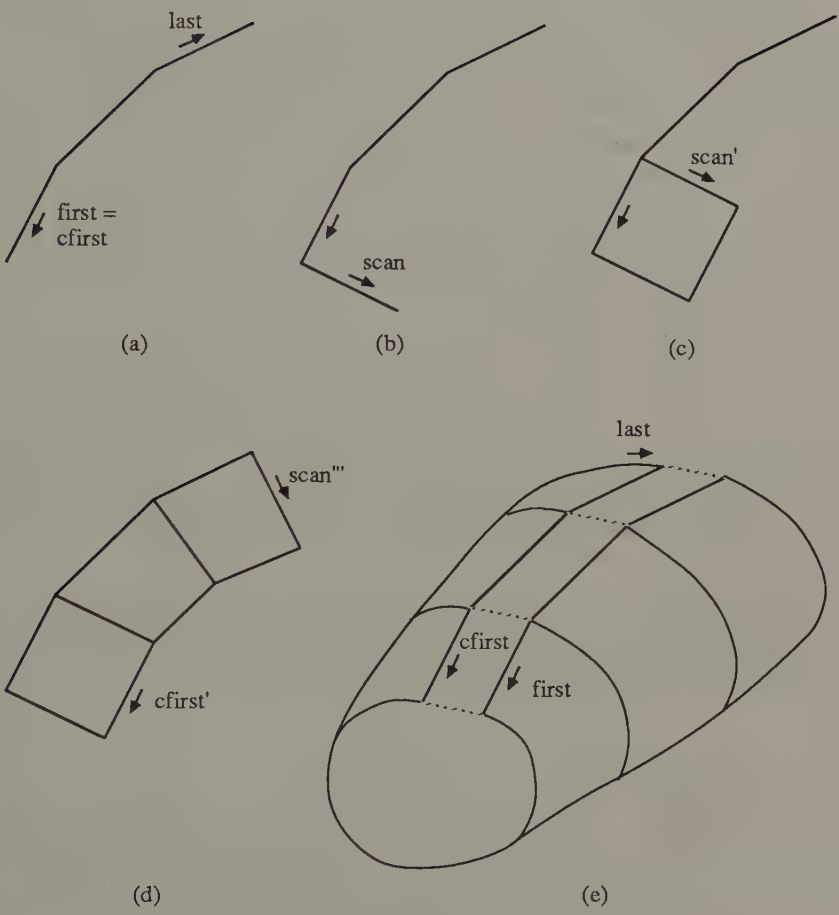



**Figure 12.1** Operation of the rotational sweeping algorithm.

ure 12.1(d), where the most recent edges created must still be joined with the edges of the original wire. This is implemented in the second phase.

The procedure of Program 12.5 implements the algorithm by first finding the head *first* of the wire; variables *cfirst* and *clast* are used to hold the leftmost edge of the current scan (see Figure 12.1(a)). The first `while`-loop controls the $nfaces - 1$ wire scans, while inner loop marches along the wire and creates the new edges; *clast* is used to control the termination of the inner loop. This leads us the situation depicted in Figure 12.2(d). In the last "mending" scan, the remaining faces and edges are created. Again, the best approach is to use low-level operators.

Based on the rotational sweeping algorithm of Program 12.5, it would seem tempting to implement a ball primitive as shown in Program 12.6. The procedure `ball` displayed works by generating a half-circle of *nver* edges, and rotating it in *nhor* steps.

Unfortunately, while the graphical result may seem correct, the algorithm of Program 12.5 will rotate also the end vertices of the arc (lying on the $x$-axis), generating two faces with *nhor* coincident vertices at the "poles" of the sphere. Such faces are likely to cause problems in numerical applications, and should be avoided. The proper generalization of the rotational sweeping algorithm to deal with the case when the wire hits the $x$-axis is left as an exercise to the reader.

Observe also that the algorithm is not capable of rotating a closed figure; i.e., solids with holes cannot be produced with it. The generalization of Program 12.5 to do this will be tackled in Section 12.5 after some necessary machinery has been developed.

## 12.4 GLUING

All algorithms of this chapter so far have been *local modifications* in that they work on one half-edge data structure instance. Another approach for solid description is the use of various sorts of *combination operations* that combine simpler solids to more complicated ones.

Prime examples of these operations are, of course, the Boolean set operations. As they (and related advanced operations) are the topic of Chapters 14 and 15, we shall here take a look into the much simpler problem of "gluing" two solids over a common face.

Let us consider the arrangement of two 6-sided cylinders, positioned right next to each other as depicted on the left side of Figure 12.2. Our aim is to glue the two cylinders into one taller solid like the one on the right side. (Observe that the cylinders meet along a pair of entirely coincident faces.)

```
void    rsweep(s, nfaces)
Solid   *s;
int     nfaces;
{
        HalfEdge        *first, *cfirst, *last, *scan;
        Face            *tailf;
        matrix          m;
        vector          v;

        getmaxnames(s);
        first = s->sfaces->floops->ledg;
        while(first->edg != first->nxt->edg) first = first->nxt;
        last = first->nxt;
        while(last->edg != last->nxt->edg) last = last->nxt;
        cfirst = first;
        matident(m);
        matrotate(m, (360.0 / nfaces), 0.0, 0.0);
        while(--nfaces)
        {
                vecmult(v, cfirst->nxt->vtx->vcoord, m);
                lmev(cfirst->nxt, cfirst->nxt, ++maxv,
                        v[0], v[1], v[2]);
                scan = cfirst->nxt;
                while(scan != last->nxt)
                {
                        vecmult(v, scan->prv->vtx->vcoord, m);
                        lmev(scan->prv, scan->prv, ++maxv,
                                v[0], v[1], v[2]);
                        lmef(scan->prv->prv, scan->nxt, ++maxf);
                        scan = mate(scan->nxt->nxt);
                }
                last = scan;
                cfirst = mate(cfirst->nxt->nxt);
        }
        tailf = lmef(cfirst->nxt, mate(first), ++maxf);
        while(cfirst != scan)
        {
                lmef(cfirst, cfirst->nxt->nxt->nxt, ++maxf);
                cfirst = mate(cfirst->prv)->prv;
        }
}
```

**Program 12.5** Rotational sweeping.

```
Solid   *ball(r, nver, nhor)
float   r;
int     nver, nhor;
{
        Solid           *s;

        s = mvfs(1, 1, -r, 0.0, 0.0);
        arc(s, f, 1, 0.0, 0.0, r, 0.0, 180.0, 0.0, nver);
        rsweep(s, nhor);
        return(s);
}
```

**Program 12.6** Ball primitive.

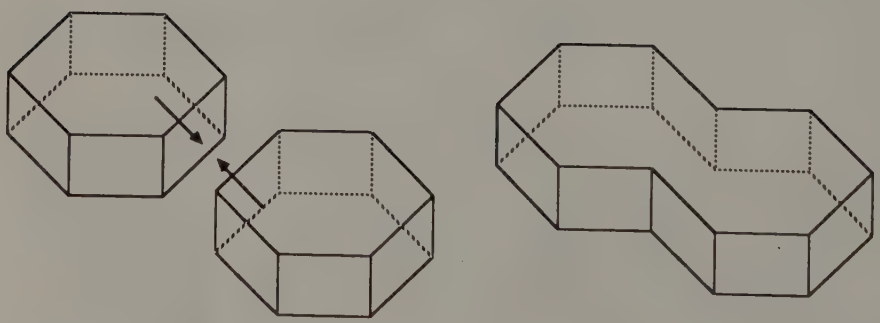


**Figure 12.2** Gluing of two cylinders.

```
void    glue(s1, s2, f1, f2)
Solid   *s1, *s2;
Face    *f1, *f2;
{
        merge(s1, s2);
        lkfmrh(f1, f2);
        loopglue(f1);
}
```

**Program 12.7** The gluing procedure.

The solution of the gluing problem splits into three parts, the combination of which gives the desired procedure of Program 12.7:

1. The merging of two half-edge data structures into one data structure that has two shells.

2. The joining of the shells. This amounts to applying a `lkfmrh` to the two glued faces.

3. The merging of coincident edges and vertices of the face resulting from `lkfmrh`.

### 12.4.1 Joining of Solids

The first step of the gluing algorithm, the procedure `merge`, joins two solids by making their all faces, edges, and vertices appear in one half-edge data structure instance. Using the procedures `dellist` and `addlist` described in Chapter 11, it is readily implemented as in Program 12.8.

### 12.4.2 Loop Gluing

After `merge`, the two glue faces can be joined with a `lkfmrh`. Here `lkfmrh` really acts as a "shell deletion" operator: it modifies the object by reducing a disconnected object to a connected one. The result of this is a face consisting of two loops, both with the identical number of vertices going pairwise through identical points. This arrangement is pictured by the plane model on the left side of Figure 12.3.

While the face satisfies the topological integrity criteria, we do not want to have such things in a "finished" model. Clearly, the desired arrangement in this case is the arrangement of the right side, where "identical" vertices

```
void    merge(s1, s2)
Solid   *s1, *s2;
{
        while(s2->sfaces)
        {
                dellist(FACE, s2->sfaces, s2);
                addlist(FACE, s2->fsaces, s1);
        }
        while(s2->sedges)
        {
                dellist(EDGE, s2->sedges, s2);
                addlist(EDGE, s2->sedges, s1);
        }
        while(s2->sverts)
        {
                dellist(VERTEX, s2->sverts, s2);
                addlist(VERTEX, s2->sverts, s1);
        }
        del(SOLID, s2, NIL);
}
```

**Program 12.8** Joining of solids.

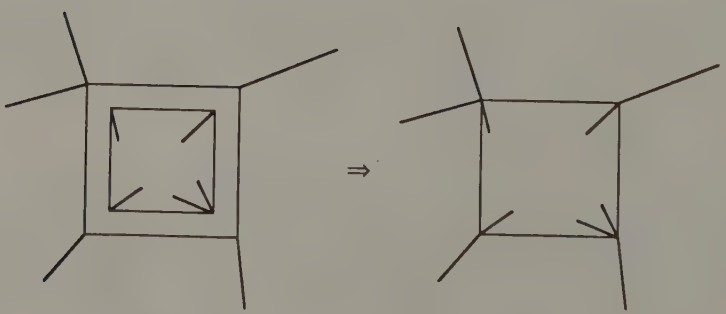


**Figure 12.3** Loop gluing.

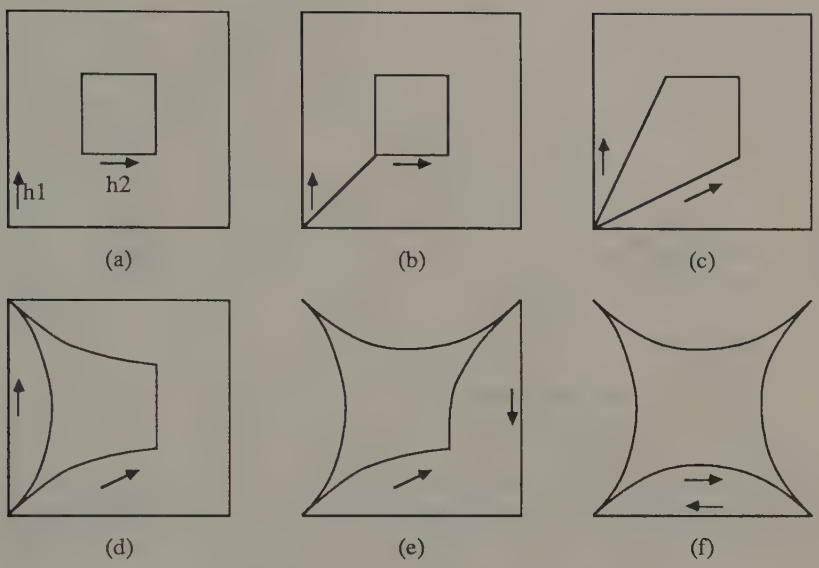



**Figure 12.4** Operation of the loop gluing algorithm.

and edges have pairwise been combined, and the "glue" face has completely disappeared.

The transformation of Figure 12.3 is the task of the *loop gluing* procedure `loopglue`. It is best implemented in terms of low-level Euler operators as in Program 12.9. The algorithm first locates a pair of halfedges *h1* and *h2* whose vertices "match," and then advances *h1* around the loop, joining edges and vertices.

The detailed operation of the algorithm is best explained with help of Figure 12.4. The first `lmekr` joins the two matching vertices with a new edge, which is immediately removed with `lkev`, leading us to the arrangement of Figure 12.4(c). The `while` loop adds an edge between the next matching vertex pair with a `lmef`, and joins them with `lkev`. One of the identical edges is removed. Observe that the temporary face with identifier $-1$ will be immediately removed by the `lkef` that follows. The loop terminates with the arrangement of Figure 12.4(f), and the last `lkef` removes the last duplicate edge.

This algorithm joins all matching pairs of vertices with new edges, and collapses the vertices together by removing the edge immediately. As these intermediate edges join two vertices with identical coordinates, they do

```
void    loopglue(fac)
Face    *fac;
{
        HalfEdge        *h1, *h2, *h1next;

        h1 = fac->floops->ledg;
        h2 = fac->floops->nextl->ledg;
        while(!match(h1, h2)) h2 = h2->nxt;
        lmekr(h1, h2);
        lkev(h1->prv, h2->prv);
        while(h1->nxt != h2)
        {
                h1next = h1->nxt;
                lmef(h1->nxt, h1->prv, -1);
                lkev(h1->nxt, mate(h1->nxt));
                lkef(mate(h1), h1);
                h1 = h1next;
        }
        lkef(mate(h1), h1);
}

int             match(h1, h2)
HalfEdge        *h1, *h2;
{
        ...
}
```

**Program 12.9** The loop gluing procedure.

```
Solid   *torus(sn, r1, r2, nf1, nf2)
Id      sn;
float   r1, r2;
int     nf1, nf2;
{
        Solid       *s;

        s = circle(sn, 0.0, r1, r2, 0.0, nf2);
        rsweep(s, nf1);
        return(s);
}
```

**Program 12.10** Torus primitive.

not quite satisfy our notion of a geometrically sensible edge. Later it will turn out, however, that various arrangements involving these *null edges* are useful when manipulating half-edge data structures with Euler operators.

## 12.5 ROTATIONAL SWEEPING REVISITED

While developed above for a rather peculiar use, `loopglue` will have many uses. The reader may be surprised to see that it is vital for generalizing the rotational sweeping algorithm as follows.

Recall that the rotational sweeping algorithm was designed so as to operate on an *open* polygonal string of edges, a wire. Hence without modifying the rotational sweeping algorithm of Program 12.5 we cannot generate a torus by the algorithm of Program 12.10 because the circle is a closed figure, i.e., a lamina with two faces.

The solution is surprisingly simple. First, we "open" the lamina at an arbitrary vertex by duplicating it with a MEV, and removing the null edge connecting the resulting two vertices with a KEF. This gives us an open wire (albeit it has a pair of vertices lying at the same point) that can be swung with the procedure `rsweep`.

Consider now the arrangement that arises after these steps (see Figure 12.5(c)). The two copies of the split vertex cause two circular faces, right on top of each other, to appear in the result of `rsweep`. This obviously is a "canonical" case where KFMRH will do the right thing by subtracting one of the faces from the other and thereby creating a hole. After that, we have a face with two exactly matching loops—i.e., exactly the arrangement where `loopglue` produces the desired answer.

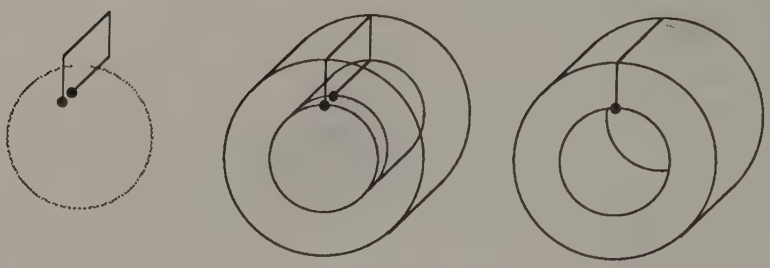


**Figure 12.5** Swinging a closed figure.

The changes needed in the procedure **rsweep** amount to inserting the code segments of Program 12.11 into the code of Program 12.5, before and after all other processing.

That is, if the solid to be swept is not a wire, it is assumed to be a lamina. The lamina is broken at an arbitrary vertex to yield a single face *headf*. After the ordinary rotational sweep, the "end" faces *headf* and *tailf* of the resulting object are merged together. (This generalization was anticipated in Program 12.5 by already setting the variable *tailf* only needed here.)

## PROBLEMS

12.1. Write a procedure for creating a *fillet arc* at a sharp corner. The procedure can have the following calling sequence:

```
int             fillet(he, radius)
HalfEdge        *he;
float           radius;
```

That is, the procedure is expected to replace the vertex **he->vtx** with an arc of the specified radius so that the arc connects tangentially with the neighboring edges (i.e., **he->prv->edg** and **he->edg**).

*Hint:* Calculate first the center and the end points of the arc. Subdivide the neighboring edges at the end points, and use a modification of the procedure **arc** of Program 12.1 to actually create the fillet arc. Thereafter, remove the unnecessary remainders of the original edges. Special cases of the operation are

(a) the angle formed by the edges is zero or 180 degrees;

(b) the radius is too large, i.e., the end points do not occur within the neighbor edges; and

```
void    rsweep(s, nfaces)
Solid   *s;
int     nfaces;
{
        ...
        HalfEdge        *h;
        Face            *headf;
        int             closed_figure;

        if(s->sfaces->nextf)  /* does the solid have > 1 faces ? */
        {
                /* assume it's a lamina */
                        closed_figure = 1;
                h = s->sfaces->floops->ledg;
                lmev(h, mate(h)->nxt, ++maxv,
                        h->vtx->vcoord[0],
                        h->vtx->vcoord[1],
                        h->vtx->vcoord[2]);
                lkef(h->prv, mate(h->prv));
                        headf = s->sfaces;
        }
        else    closed_figure = 0;              /* no, it's a wire */
        ...
        if(closed_figure == 1)
        {
                lkfmrh(headf, tailf);
                loopglue(headf);
        }
}
```

**Program 12.11** Rotational sweeping of closed figures.

(c) the fillet arc "eats up" one or both of the neighbor edges, i.e., the fillet radius is the largest permitted by item (b) above.

12.2. Generalize the rotational sweeping algorithm of Program 12.5 to work correctly in the case that one or both end points of the swept wire lie on the $x$-axis.

*Hint:* Write an algorithm that reduces a face to a single vertex, and apply it to the faces *headf* and *tailf* of Program 12.11.

12.3. Generalize the modified rotational sweeping algorithm of Program 12.11 to work correctly with closed figures that have an edge lying on $x$-axis.

12.4. Write the procedure

```
int     unsweep(f)
Face    *f;
```

that will "flatten" a solid generated by sweeping the face $f$. (Observe that $f$ will appear as the "top" face of the swept part.)

## BIBLIOGRAPHIC NOTES

The story of implementing various higher-level tools on the top of Euler operators goes on. Solid modelers BUILD-2 [18] and ROMULUS include a wide collection of "local manipulations" (various forms of sweeping, tweaking, bending, etc.) implemented on top of Euler operators. MODIF [25] allows the user to employ Euler operators in quite a direct fashion to define a polyhedral "initial" solid as the initial model for generating curved surfaces. In similar vein, also ROMULUS supports "drafting"-type operations that give the user considerable freedom as for creating geometric objects on top of Euler operators.

# Chapter 13

# GEOMETRIC ALGORITHMS

With the tools of Chapter 12 we can generate models of simple solids. We now shift our attention to algorithms that can perform simple geometric calculations on them. These geometric primitives will be used extensively in the more advanced algorithms of GWB to be discussed in the subsequent chapters.

## 13.1 FACE EQUATIONS

Face nodes of the half-edge data structure include storage for plane equations, but no Euler operator attempts to put any data into it. The procedure for calculating the face equations is hence the first geometric tool we need.

An efficient and numerically robust way to calculate the face equation is the method of Newell [114]. Let the boundary of the face consist of $n$ vertices $v_1, ..., v_n$, and let the coordinates of $v_i$ be $x_i$, $y_i$, and $z_i$. Then the coefficients $a$, $b$, and $c$ of the plane equation

$$[a\ b\ c\ d] \cdot [x\ y\ z\ 1]^T = 0$$

can be calculated by

$$\begin{aligned} a &= \sum_{i=1}^{n} (y_i - y_{i+1})(z_i + z_{i+1}) \\ b &= \sum_{i=1}^{n} (z_i - z_{i+1})(x_i + x_{i+1}) \end{aligned} \tag{13.1}$$

$$c = \sum_{i=1}^{n} (x_i - x_{i+1})(y_i + y_{i+1})$$

where index addition is modulo $n$.

The remaining coefficient is calculated by selecting a point on the plane of the face, e.g., the point $(x_{av}\ y_{av}\ z_{av})$ calculated as the arithmetic average of all its vertices. Then the selection

$$d = -[x_{av}\ y_{av}\ z_{av}] \cdot [a\ b\ c]^T \tag{13.2}$$

makes sure that $(x_{av}\ y_{av}\ z_{av})$ will satisfy the face equation.

Procedure **faceeq** of Program 13.1 implements this method. It calculates the face equation based on one loop of the face, normally the "outer" one. As the equation can be scaled by a constant without altering the plane it describes, the procedure normalizes the equation so that $[a\ b\ c]$ becomes a vector of unit length. The special case of $[a\ b\ c]$ becoming a null vector will occur if a loop of zero area is given to the procedure—e.g., an open wire.

Many geometric algorithms that we shall examine will operate by projecting the scene on one of the coordinate planes. The coordinate plane must, of course, be selected so as to avoid singularities. For instance, we must beware of projecting a face whose normal is [0 0 1] onto coordinate planes other than the $xy$-plane. This leads us to the procedure **dropcoord** that selects the coordinate "dropped" by determining which of the coefficients $[a\ b\ c]$ of a plane equation has the largest absolute value. The procedure is assumed to return one of the values $X$, $Y$, or $Z$ defined as literals. Its implementation is left to the reader.

## 13.2 CONTAINMENT AND INTERSECTION ALGORITHMS

The geometric primitives we need break into two classes, namely those that test for *containment* of two geometric entities, and those that test for *intersection* of geometric entities. These procedures form a natural hierarchy, as we shall see.

### 13.2.1 Vertex Equality

We have already met a case where a containment procedure is needed: the loop gluing algorithm **loopglue** of Program 12.9 uses the procedure `match(h1, h2)` to test whether the vertices of halfedges *h1* and *h2* occupy the same point.

```
int     faceeq(l, eq)
Loop    *l;
vector  eq;
{
        HalfEdge        *he;
        double          a, b, c, norm;
        double          xi, yi, zi, xj, yj, zj, xc, yc, zc;
        int             len;

        a = b = c = xc = yc = zc = 0.0;
        len = 0;
        he = l->ledg;
        do
        {
                xi = he->vtx->vcoord[0];
                yi = he->vtx->vcoord[1];
                zi = he->vtx->vcoord[2];
                xj = he->nxt->vtx->vcoord[0];
                yj = he->nxt->vtx->vcoord[1];
                zj = he->nxt->vtx->vcoord[2];
                a += (yi - yj) * (zi + zj);
                b += (zi - zj) * (xi + xj);
                c += (xi - xj) * (yi + yj);
                xc += xi;
                yc += yi;
                zc += zi;
                len++;
        }
        while((he = he->nxt) != l->ledg);
        if((norm = sqrt(a*a + b*b + c*c)) != 0.0)
        {
                eq[0] = a / norm;
                eq[1] = b / norm;
                eq[2] = c / norm;
                eq[3] = (eq[0]*xc + eq[1]*yc + eq[2]*zc) / (-len);
                return(SUCCESS);
        }
        else
        {
                printf("faceeq: null face %d\n", l->lface->faceno);
                return(ERROR);
        }
}
```

**Program 13.1** Evaluation of face equations.

```
int     contvv(v1, v2)
Vertex  *v1, *v2;
{
        double  dx, dy, dz;
        double  diff;

        dx = v1->vcoord[0] - v2->vcoord[0];
        dy = v1->vcoord[0] - v2->vcoord[0];
        dz = v1->vcoord[0] - v2->vcoord[0];

        diff = dx * dx + dy * dy + dz * dz;
        return(comp(diff, 0.0, EPS*EPS) == 0);
}

int     comp(a, b, tol)
double  a, b, tol;
{
        double  delta;

        delta = fabs(a - b);
        if(delta < tol) return(0);
        else if(a > b)  return(1);
        else            return(-1);
}
```

**Program 13.2** Vertex equality.

This vertex coincidence test is the task of `contvv`, the simplest primitive we are going to need. Albeit simple, it must be implemented carefully while taking small numerical errors into account. The implementation of Program 13.2 uses the Euclidean distance metric to this end. If the distance is less than some small value, the vertices are considered coincident.

Procedure `comp` is used to perform the comparison of the calculated distance to zero. It performs a three-way comparison between its two first arguments: if their absolute difference is less than the third argument, it returns a zero, and otherwise $-1$ or 1 for first argument less than second and vice versa. We shall use `comp` in all algorithms where numerical accuracy may be a problem.

The selection of the value for *EPS* must be made carefully on the basis of the actual overall dimensions of the object being processed. If single-precision arithmetic is used, a value of one millionth of the largest dimension of the object can serve as a starting point. Another, more advanced

approach to selecting the tolerances is to actually measure the "fuzziness" of the object being processed, e.g., the largest absolute distance of a vertex from a face it is supposed to lie on, and use some multiple of that value as the base tolerance.

### 13.2.2 Vertex-Edge Containment

Let us now turn our attention to a slightly more complicated problem, the vertex-edge containment: given a vertex $v$ and an edge $e$, test whether $v$ lies on $e$ or not. Despite this simple specification, there are design decisions to be made. How are the edges represented? How is the intersection (if any) represented?

We choose to describe "edges" for this algorithm in terms of two vertices $v_1$ and $v_2$. They may or may not correspond with a pair of vertices actually connected with an edge. The edge is considered as *directed* from $v_1$ to $v_2$, which gives each point $p$ of the line of $e$ the parametric representation

$$p(t) = v_1 + t(v_2 - v_1).$$

Points $p(t)$ where $0 < t < 1$ form the interior of the edge. In particular, points $p(t)$ with parameter values $t = 0$ and $t = 1$ correspond with the endpoints $v_1$ and $v_2$ and form the boundary of the edge.

To test the inclusion of the test vertex *v3* on the edge between vertices *v1* and *v2*, the implementation of Program 13.3 determines first the intersection of *v3* and the infinite line through *v1* and *v2* by projecting *v3* on the line and testing the equality of *v3* and the projected vertex *testv* by means of contvv. For a detected intersection, the parameter $t$ is reported, and the containment procedure simply tests whether $t$ is between 0 and 1.

When considered in separation, it would seem better to let Program 13.3 provide a different answer in the case that $t$ equals 0.0 or 1.0, denoting that the test vertex is on the boundary of the edge. But this test is much better handled by contvv, and the way we shall use the two procedures makes this refinement unnecessary.

### 13.2.3 Vertex-Loop Containment

The next task we shall look at is an algorithm for testing the containment of a vertex in a loop, a task often called the "point-in-polygon" problem: given a loop $l$ and a vertex $v$, determine whether $v$ is in the interior of $l$ or not.

In our case, loops are three-dimensional objects, and the containment does not seem well-defined if $v$ is not coplanar with $l$. However, in GWB, the

```
int     intrev(v1, v2, v3, t)
Vertex  *v1, *v2, *v3;
double  *t;
{
        Vertex  testv;
        float   r1[3], r2[3], r1r1, tprime;

        r1[0] = v2->vcoord[0] - v1->vcoord[0];
        r1[1] = v2->vcoord[1] - v1->vcoord[1];
        r1[2] = v2->vcoord[2] - v1->vcoord[2];

        r1r1 = dot(r1, r1);
        if(r1r1 < EPS*EPS)
        {
                *t = 0.0;
                return(contvv(v1, v3));
        }
        else
        {
                r2[0] = v3->vcoord[0] - v1->vcoord[0];
                r2[1] = v3->vcoord[1] - v1->vcoord[1];
                r2[2] = v3->vcoord[2] - v1->vcoord[2];
                tprime = dot(r1, r2) / r1r1;
                testv.vcoord[0] = v1->vcoord[0] + tprime * r1[0];
                testv.vcoord[1] = v1->vcoord[1] + tprime * r1[1];
                testv.vcoord[2] = v1->vcoord[2] + tprime * r1[2];
                *t = tprime;
                return(contvv(&testv, v3));
        }
}

int     contev(v1, v2, v3)
Vertex  *v1, *v2, *v3;
{
        double  t;

        if(intrev(v1, v2, v3, &t))
                if(t >= (-EPS) && t <= (1.0+EPS))
                        return(1);
        return(0);
}
```

**Program 13.3** Vertex-edge intersection.

```
HalfEdge        *hithe;
Vertex          *hitvertex;

int     bndrlv(l, v)
Loop    *l;
Vertex  *v;
{
        HalfEdge        *he;

         he = l->ledg;
        do
        {
                if(contvv(he->vtx, v))
                {
                        hitvertex = he->vtx;
                        hithe = NIL;
                        return(3);
                }
        }
        while((he = he->nxt) != l->ledg);
        he = l->ledg;
        do
        {
                if(contev(he->vtx, he->nxt->vtx, v))
                {
                        hitvertex = NIL;
                        hithe = he;
                        return(2);
                }
        }
        while((he = he->nxt) != l->ledg);
}
```

**Program 13.4** Boundary cases.

procedure will be used in situations where the test vertex is approximately on the plane of the loop by construction. This makes it possible for our algorithm to work in one of the coordinate planes. The calling procedure specifies the projection plane in terms of a parameter *drop* that identifies the coordinate axis to be ignored; *drop* can be calculated with `dropcoord`.

Often it is important to separate cases where the test vertex is coincident with a vertex of the loop or lies on an edge of the loop. This initial test is performed in Program 13.4 by using the simpler procedures developed so far.

Observe that the procedure `bndrlv` first checks the test vertex against *all* vertices of the loop before making any edge containment tests. This makes certain that the stronger result of vertex coincidence is noticed. Besides returning a code indicating the result of the test, the algorithm also sets the global variables *hitedge* and *hitvertex* to point at the detected intersecting object. We shall make use of this feature later in Chapter 15.

The actual vertex-loop containment algorithm of Program 13.5 first calls the procedure for checking boundary intersections so as to separate out any special cases—again, robustness requires that the stronger test is performed first. If the test vertex does not lie on the boundary of the loop, a containment test based on the "ray-firing" technique is performed. That is, from the test vertex $v$ a ray is sent to infinity, and *count*, the number of intersections between the ray and the edges of the reference loop $l$ is calculated. If *count* is even, $v$ is outside $l$, and otherwise within it. The procedure `int2ee` for determining the intersection of two lines is presented in the next section.

The main difficulty of this method is to properly treat cases where the ray hits a vertex or coincides with an edge of the polygon. In Program 13.5, the ray is initially selected so as to go from $v$ through the center point of the first edge of the loop. If this choice leads into a difficulty, the algorithm "panics" and selects another ray. While perhaps lacking esthetic appeal (and efficiency), this technique is quite tolerant to numerical inaccuracies.

The loop containment procedure can be made somewhat more efficient by including a "box test": if the vertex falls outside the bounding box of the loop, it can immediately be decided to be out.

On top of `contlv`, an analogous procedure `contfv` for testing the containment of a vertex in a face can be built. The procedure can work by checking the inner loops only if the vertex is found to be within the outer loop. We leave its implementation as an exercise for the reader, but shall feel free to use it in the later chapters.

```
int     contlv(l, v, drop)
Loop    *l;
Vertex  *v;
int     drop;
{
    HalfEdge        *he1, *he2;
    Vertex          *v1, *v2, aux;
    double          t1, t2;
    int             count, intr, c1, c2;

    if((intr = bndrlv(l, v)) > 0) return(intr);
    he2 = l->ledg;
retry:
    v1 = he2->vtx;
    v2 = he2->nxt->vtx;
    aux.vcoord[0] = (v1->vcoord[0] + v2->vcoord[0]) / 2.0;
    aux.vcoord[1] = (v1->vcoord[1] + v2->vcoord[1]) / 2.0;
    aux.vcoord[2] = (v1->vcoord[2] + v2->vcoord[2]) / 2.0;
    he1 = l->ledg;
    count = 0;
    do
    {
        intr = int2ee(v, &aux, v1, v2, drop, &t1, &t2);
        if(intr == 1)
        {
            c1 = comp(t2, 0.0, EPS);
            c2 = comp(t2, 1.0, EPS);
            if(c1 == 0 || c2 == 0)
            {
                he2 = he2->nxt;
                if(he2 == l->ledg) return(ERROR);
                goto retry;
            }
            if(c1 == 1 && c2 == -1)
                if(t1 >= 0.0) count++;
        }
    }
    while((he1 = he1->nxt) != l->ledg);
    count = count % 2;
    return(count);
}
```

**Program 13.5** Vertex-loop containment.

### 13.2.4 Line Intersection

What remains is the procedure `int2ee`, whose task is calculate the intersection of two lines on the coordinate plane determined by *drop* as above. Similarly to `intrev`, it returns parameter values that correspond with the intersection point.

The implementation of Program 13.6 works by solving the two linear equations derived from

$$v_1 + t_1(v_2 - v_1) = v_3 + t_2(v_4 - v_3)$$

directly for $t_1$ and $t_2$. The calling procedure can determine the existence of an intersection by examining whether $t_1$ and $t_2$ occur in the range $[0, 1]$.

## 13.3 INTEGRAL PROPERTIES

Let us now take a look at the problem of evaluating elementary geometric properties of solids. We concentrate on calculating the volume and the surface area of a solid; nevertheless, other integral properties could be calculated with very similar procedures.

### 13.3.1 Volume

When alternative approaches to algorithms for evaluating integral properties of boundary models were discussed on a general level in Chapter 6, it was noted that consistently oriented polyhedral models can be analyzed by special algorithms tailored to take advantage of the consistent orientation.

The compact algorithm of Program 13.7 implements the method mentioned in Chapter 6 for evaluating the volume of a polyhedron. It works by summing the signed volumes of all tetrahedra formed by vertices of each consecutive pair of edges and a fixed point at the origin. In the solid is far from the origin, the accuracy can be improved drastically by selecting a point inside the solid as the fixed point.

### 13.3.2 Surface Area

The primitive operation for the evaluation of surface area of a solid is the calculation of the area of a loop, and we shall concentrate on it only. The development of procedures that can properly sum loop areas to give the total surface area of a solid is straightforward and left to the reader.

The simplest algorithm for evaluating the area of a loop uses the loop normal vector *norm*, assumed to be a unit vector. We select a fixed vertex

```
int     int2ee(v1, v2, v3, v4, drop, t1, t2)
Vertex  *v1, *v2, *v3, *v4;
int     drop;
double  *t1, *t2;
{
        double  d, a1, a2, b1, b2, c1, c2;

        switch(drop)
        {
                case X:
                        a1 = v2->vcoord[1] - v1->vcoord[1];
                        a2 = v2->vcoord[2] - v1->vcoord[2];
                        b1 = v3->vcoord[1] - v4->vcoord[1];
                        b2 = v3->vcoord[2] - v4->vcoord[2];
                        c1 = v1->vcoord[1] - v3->vcoord[1];
                        c2 = v1->vcoord[2] - v3->vcoord[2];
                        break;
                case Y:
                        a1 = v2->vcoord[0] - v1->vcoord[0];
                        a2 = v2->vcoord[2] - v1->vcoord[2];
                        b1 = v3->vcoord[0] - v4->vcoord[0];
                        b2 = v3->vcoord[2] - v4->vcoord[2];
                        c1 = v1->vcoord[0] - v3->vcoord[0];
                        c2 = v1->vcoord[2] - v3->vcoord[2];
                        break;
                case Z:
                        a1 = v2->vcoord[0] - v1->vcoord[0];
                        a2 = v2->vcoord[1] - v1->vcoord[1];
                        b1 = v3->vcoord[0] - v4->vcoord[0];
                        b2 = v3->vcoord[1] - v4->vcoord[1];
                        c1 = v1->vcoord[0] - v3->vcoord[0];
                        c2 = v1->vcoord[1] - v3->vcoord[1];
                        break;
        }
        if(comp((d = a1*b2 - a2*b1), 0.0, EPS) == 0)
                return(0);
        *t1 = (c2*b1 - c1*b2) / d;
        *t2 = (a2*c1 - a1*c2) / d;
        return(1);
}
```

**Program 13.6** Line intersection.

```
double svolume(s)
Solid  *s;
{
        Face            *f;
        Loop            *l;
        HalfEdge        *he1, *he2;
        vector          c;
        double          res;

        res = 0.0;
        f = s->sfaces;
        while(f)
        {
                l = f->floops;
                while(l)
                {
                        he1 = l->ledg;
                        he2 = he1->nxt;
                        do
                        {
                                cross(c, he1->vtx->vcoord,
                                        he2->vtx->vcoord);
                                res += dot(he2->nxt->vtx->vcoord, c);
                        }
                        while((he2 = he2->nxt) != he1);
                        l = l->nextl;
                }
                f = f->nextf;
        }
        return(res / 6.0);
}
```

**Program 13.7** Volume evaluation.

$p_1$ (say, the first vertex) from the loop, and calculate the contribution of each edge $p_i, p_{i+1}$ to the area as the cross product

$$v_i = (p_i - p_1) \times (p_{i+1} - p_1). \tag{13.3}$$

Then the total area is given by

$$v = \frac{1}{2} \times \sum norm \cdot v_i. \tag{13.4}$$

The procedure `larea` of Program 13.8 implements this algorithm. It could work somewhat more efficiently by means of fetching face equations from `l->lface->feq` instead of re-evaluating them.

## PROBLEMS

13.1. The accuracy of the face equation calculated with the procedure `faceeq` of Program 13.1 can be considerably enhanced by translating the face at origin. Modify Program 13.1 so as to implement this.

13.2. Write the procedure `dropcoord` that has the following calling sequence:

```
int     dropcoord(v)
vector  v;
```

The procedure determines which of the three first coefficients of the vector $v$ has the largest absolute value, and returns one of the literals $X$, $Y$, or $Z$, respectively.

13.3. Implement the procedure `contfv` that has the following calling sequence:

```
int     contfv(f, v)
Face    *f;
Vertex  *v;
```

The procedure should test for the inclusion of the vertex $v$ in the face $f$ and return similar values as `contlv` of Program 13.5.

*Hint:* Use `contlv`.

13.4. Implement the "shorthand" procedure `contfp` that has the following calling sequence:

```
int     contfp(f, x, y, z)
Face    *f;
float   x, y, z;
```

The procedure should test for the inclusion of the point $(x\ y\ z)$ in the face $f$ and return similar values as `contlv` of Program 13.5.

```
double  larea(l)
Loop    *l;
{
        HalfEdge        *he;
        Vertex          *v1, *v2, *v3;
        vector          aa, bb, cc, dd, norm;

        dd[0] = dd[1] = dd[2] = 0.0;
        faceeq(l, norm);
        he = l->ledg;
        v1 = he->vtx;
        he = he->nxt;
        do
        {
                v2 = he->vtx;
                v3 = he->nxt->vtx;
                aa[0] = v2->vcoord[0] - v1->vcoord[0];
                aa[1] = v2->vcoord[1] - v1->vcoord[1];
                aa[2] = v2->vcoord[2] - v1->vcoord[2];
                bb[0] = v3->vcoord[0] - v1->vcoord[0];
                bb[1] = v3->vcoord[1] - v1->vcoord[1];
                bb[2] = v3->vcoord[2] - v1->vcoord[2];
                cross(cc, aa, bb);
                dd[0] += cc[0];
                dd[1] += cc[1];
                dd[2] += cc[2];
        }
        while((he = he->nxt) != l->ledg);
        return(0.5 * dot(norm, dd));
}
```

**Program 13.8** Loop area evaluation.

13.5. Write the procedure

```
void    laringmv(f1, f2)
Face    *f1, *f2;
```

that moves those rings of *f1* that do not lie within its outer loop to *f2*. The procedure will be used in the next chapter to make sure that after a face has been divided by a MEF, all loops will end up in the correct halves.

13.6. Write the procedure

```
int             checkwideness(he)
HalfEdge        *he;
```

that checks whether the edges `he->prv->edg` and `he->edg` make a convex (less than 180 degrees) or concave (larger than 180 degrees) angle. In the first case, the procedure should return 1, and 0 otherwise.

*Hint:* Compare the cross product of the direction vectors of the edges with the face normal. Be sure to treat the case that the edges are colinear.

13.7. Modify procedure `svolume` of Program 13.7 to use a vertex inside the solid as the fixed point. Compare experimentally the accuracies of the two versions of the algorithm.

13.8. Based on the loop are evaluation procedure `larea` of Program 13.8, write the corresponding procedures for the area of a face and of a solid.

13.9. Write the procedures

```
void    strans(s, dx, dy, dz)
Solid   *s;
float   dx, dy, dz;

void    srotat(s, rx, ry, rz)
Solid   *s;
float   rx, ry, rz;

void    stransf(s, m)
Solid   *s;
Matrix  m;
```

that perform a translation, a rotation, and a general transformation on the solid *s*.

# Chapter 14

# A SPLITTING ALGORITHM

The tools for boundary modeling presented in the preceding chapters form the basis of yet higher-level tools. This chapter describes an algorithm for *splitting* a solid with a plane. While being a useful modeling tool by itself, it also forms a good introduction to the still more advanced subject of Boolean set operations, the topic of the next chapter.

## 14.1 THE PROBLEM

We shall consider an algorithm for solving the *splitting problem*: Given a solid $S$, and a splitting plane $SP$, calculate two solids *Above* and *Below* that represent the parts of $S$ that lie on the positive and on the negative side of $SP$, respectively. If $S$ lies completely on one side of $SP$, the solid representing the other side is empty.

Observe that this problem definition requires that the algorithm is *closed*: its results must be "solids" themselves, and hence the algorithm (and any other algorithms that work for solids) must be repeatedly applicable to its results. This means that while a splitting algorithm certainly would be useful for generating the "section views" often encountered in engineering drawings (see Figure 14.1), it is not restricted to mere visualization, but can be used as a real modeling tool.

More precisely, we shall assume that $S$ is in all respects a well-formed solid represented with a correct half-edge data structure. In particular, $S$ must be nonself-intersecting, and all of its faces, edges, and vertices must have correct geometries (up to reasonable floating-point precision).

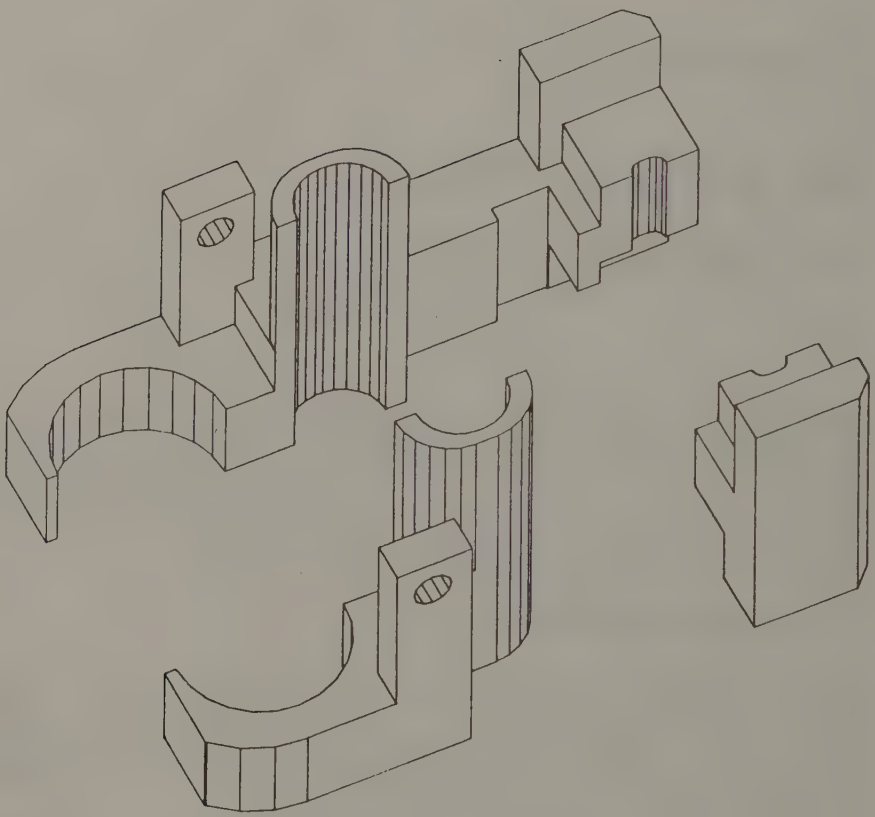


**Figure 14.1** A split solid.

If so, we require that the results of a splitting algorithm have the same correctness characteristics.

To satisfy the problem statement, a splitting algorithm must be *general*, i.e., it must work correctly for all possible geometric configurations of solids and planes. The algorithm should also be *robust*, i.e., capable of working correctly even if small numerical inaccuracies are present in the original solid $S$.

In the following, we shall develop a splitting algorithm with these characteristics. In particular, the algorithm will be completely general up to the limits permitted by computational accuracy.

## 14.2 OVERVIEW

Let us first outline the splitting algorithm at a verbal level, and introduce some terminology that will be used also in the next chapter.

Obviously, if the solid $S$ to be split and the splitting plane $SP$ intersect at all, they do so along one or more cross section polygons or just *section polygons*. For instance, the object of Figure 14.1 has five section polygons.

The splitting algorithm is expected to divide $S$ along the section polygon(s) in the two parts *Above* and *Below*. This can be achieved by inserting a pair of *section faces* along each section polygon, one in the part *Above* and the other in the part *Below*. (In engineering drawings, section faces are usually shown crosshatched.)

In the following, we shall outline this computation with help of the sample case shown in two views in Figure 14.2. In 14.2(a), the section polygons are drawn in dashed lines; 14.2(b) shows the expected result. Observe that the part *Above* in this case is a disconnected solid, which is a perfectly acceptable result of the algorithm. Observe also that several edges and vertices of the solid are coplanar with the splitting plane. (That the top surface of *Below* consists of three coplanar faces is an artifact of the splitting algorithm to be discussed.)

### 14.2.1 Reduction Step

Our plan for the splitting algorithm is to *reduce* the overall problem into a collection of subproblems that can be handled independently of each other. Because we aim at a simple solution that does not involve a large case analysis, we shall select the most primitive kind of a subproblem possible, namely the processing a vertex of the solid $S$ lying on the splitting plane $SP$. For instance, instead of processing an edge lying on $SP$ directly as one

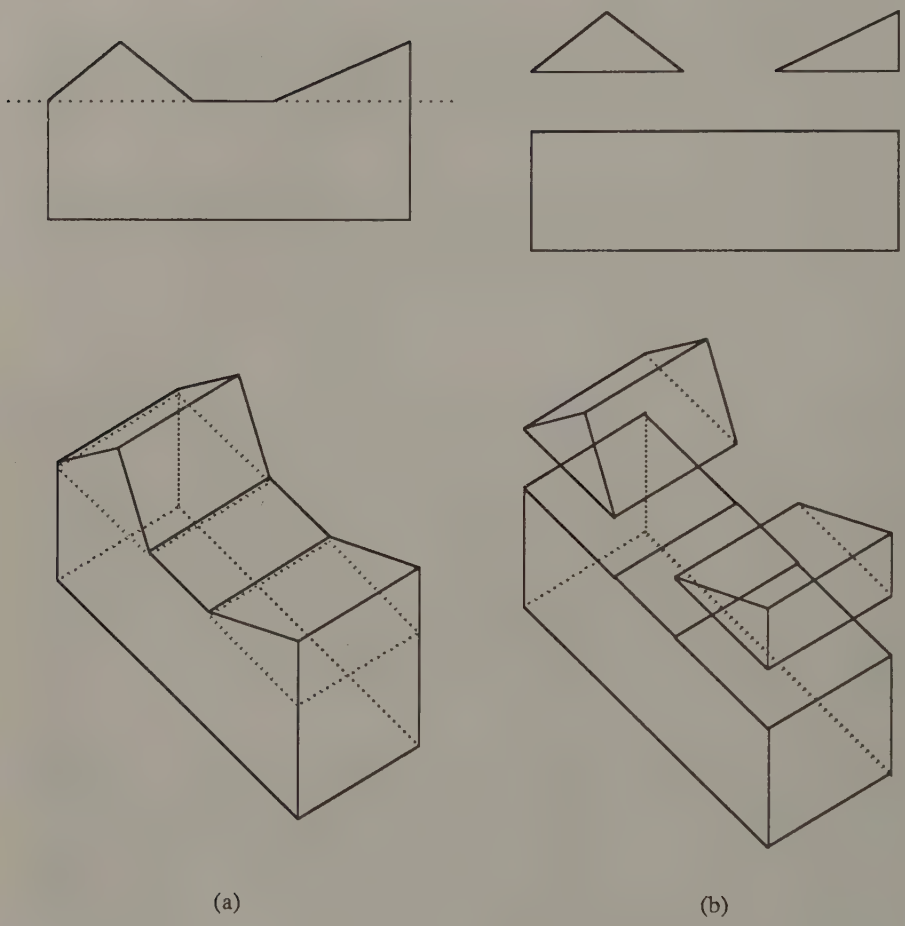



**Figure 14.2** A sample splitting problem.

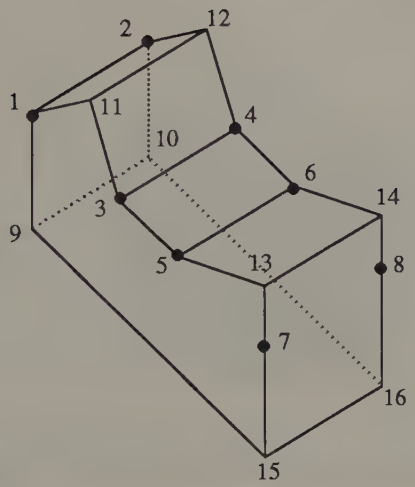



**Figure 14.3** Set of coplanar vertices for the sample splitting problem.

particular case of some case analysis, we handle it by processing its two vertices appropriately.

This reduction is the task of the first step of the algorithm. It can be described as follows:

1. Locate all edges of $S$ that intersect $SP$ properly, i.e., whose vertices lie in different sides of $SP$. Subdivide each such edge with a new vertex at its intersection point with $SP$.

2. Locate all vertices of $S$ that lie on $SP$, and store them for further processing. If no such vertices are found, we are done: the solid does not intersect $SP$ at all. Of course, the resulting set of vertices includes at least all vertices inserted during step 1.

The set of vertices generated in the reduction step of the sample splitting problem of Figure 14.2 is shown in Figure 14.3. For reference, these vertices are labeled with integers 1–8. Labels 9–16 denote the other vertices of $S$. Observe that vertices 7 and 8 were inserted into $S$ by subdividing an edge at its intersection point with $SP$.

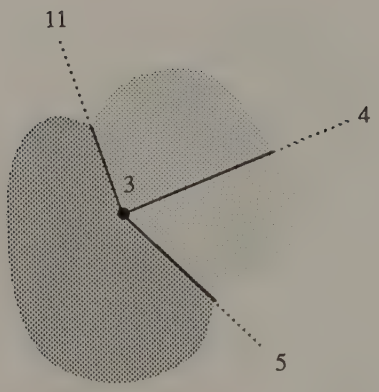



**Figure 14.4** Vertex neighborhood of vertex 3.

### 14.2.2 Vertex Neighborhood Classification

By virtue of the reduction step, we now have only one kind of a basic problem to worry about, namely the processing of a vertex lying on *SP*. Of course, this computation must be general enough to cover all possible situations.

To achieve the desired generality with a simple algorithm, is is again useful to split the processing into more primitive subproblems. Let us introduce the term *vertex neighborhood* to denote the ordered cycle of edges and faces around a vertex. For instance, the vertex neighborhood of vertex 3 of the sample problem is depicted in Figure 14.4. This neighborhood consists of the (half-)edges (3,11), (3,4) and (3,5); the vertices 11, 4, and 5 are termed the *final vertices* of the edges of the vertex neighborhood, and vertex 3 is the *base vertex* of the neighborhood. We shall use the term *sector* to denote the part of a face immediately adjacent to a vertex; in Figure 14.4, the neighborhood includes sectors (11,3,4), (4,3,5), and (5,3,11).

To solve the splitting problem, each coplanar vertex will be processed with an algorithm called here the *vertex neighborhood classifier*. The classifier is expected to locate all sectors that intersect *SP*, and to subdivide the neighborhood into parts that are wholly on one side of *SP*.

The subdivision of a neighborhood is effected by inserting new edges of zero length into it; these *null edges* will be removed in later steps of the splitting algorithm. For instance, the result of the processing of vertex 3 is

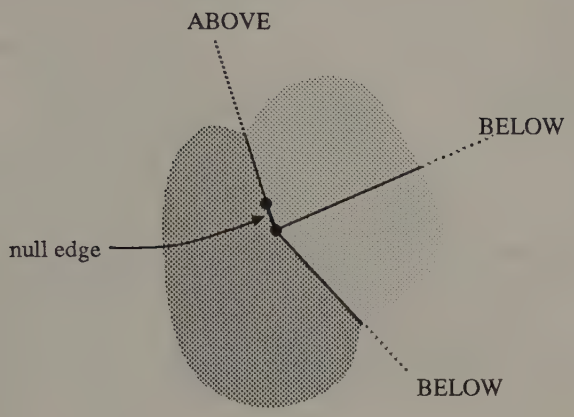



**Figure 14.5** Result of vertex neighborhood classification of vertex 3.

shown in Figure 14.5. In this case, the null edge separates the edge (3,11) (which will appear in *Above*) from the edges (3,4) and (3,5) (which will appear in *Below*).

### 14.2.3 Computation of the Result

After all coplanar vertices have been processed with the vertex neighborhood classifier, the section polygons can be constructed by combining the vertices of a null edges in a proper order, and removing the null edges themselves.

For the sample case, this process is depicted in Figure 14.6. Observe that we can choose the topological orientation of all null edges so that they will be positively oriented from *Below* to *Above*. As will be seen, this leads to a relatively simple iterative algorithm that marches around a section polygon, and combines null edges with each other. After the joining step, it is straightforward to insert the section faces, and to rip *Above* and *Below* apart.

## 14.3 OUTLINE OF THE ALGORITHM

An outline of the splitting algorithm is given in Program 14.1. In addition to the half-edge data structure, the algorithm uses three arrays for representing the "set of ON-vertices" *soov*, the "set of null edges" *sone*, and the "set of null faces" *sonf* that will be used to store the section faces. These

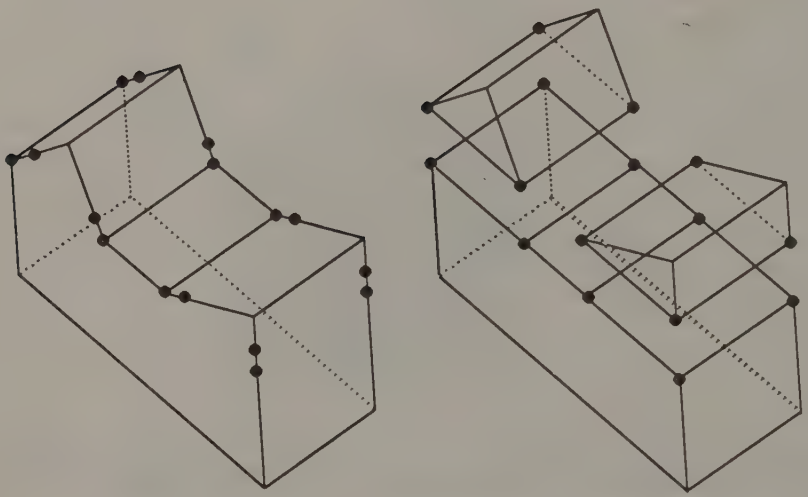


**Figure 14.6** Computation of the result.

arrays are assumed to be global information available to all parts of the splitting algorithm.

The splitting plane *SP* is represented simply by a vector whose components give the four coefficients of the plane equation. For initialization, the algorithm evaluates all face equations of the solid, and sets up the global variables *maxv* and *maxf* for generating unique identifiers in later parts. The remaining procedures will be elaborated in following sections.

## 14.4 REDUCTION STEP

Let us first consider the procedure **splitgenerate** which is responsible for generating the set of coplanar vertices *soov* for the later steps. In particular, it must be capable of subdividing all edges of $S$ that intersect $SP$ at their intersection points.

Obviously, an edge $e$ of $S$ intersects the splitting plane $SP$ if and only if its end points $p_1$ and $p_2$ lie on opposite sides of the plane. Hence **splitgenerate** can work by evaluating the signed distances $d_1$ and $d_2$ of $p_1$ and $p_2$. If either distance is (approximately) zero, the corresponding vertex is on $SP$, and can immediately be inserted to the result set. Otherwise, if the signs of the distances differ, $e$ must intersect $SP$, and the point

```
Vertex  *soov[MAXONVERTICES];
int     nvtx;
Edge    *sone[MAXNULLEDGES];
int     nedg;
Face    *sonf[MAXNULLFACES*2];
int     nfac;

void    split(S, SP, Above, Below)
Solid   *S, **Above, **Below;
vector  SP;
{
        Face    *f;

        for(f = S->sfaces; f != NIL; f = f->nextf)
                faceeq(f->flout, f->feq);
        getmaxnames(S);
        splitgenerate(S, SP);
        splitclassify(SP);
        if(nedg == 0)
        {
                printf("split: no intersections found\n");
                return;
        }
        splitconnect();
        splitfinish(S, Above, Below);
}
```

**Program 14.1** Outline of the splitting algorithm.

of intersection $p_{int}$ can be calculated as

$$p_{int} = p_1 + \frac{d_1}{d_1 - d_2}(p_2 - p_1).$$

Procedure **splitgenerate** is implemented in Program 14.2. The algorithm scans all edges of $s$ and calculates the distances *d1* and *d2* by means of the procedure **dist** not elaborated here. If the distances are nonzero and of different sign, the edge is subdivided, and the new vertex inserted to the result set. Otherwise, if either of the distances is zero, the correponding vertex is stored immediately. Of course, the comparisons with zero must be done with the procedure **comp**, which was introduced in Chapter 13.

Obviously, the algorithm will encounter each coplanar vertex several times. It could be made more efficient by calculating the signed distance of each vertex just once, and storing these data into an auxiliary array.

## 14.5 VERTEX NEIGHBORHOOD CLASSIFIER

Next, let us consider the implementation of the vertex neighborhood classifier **splitclassify**.

### 14.5.1 Reclassification Rules

The classifier must be capable of treating properly all kinds of vertex neighborhoods, including ones with sectors or edges that are coplanar with the splitting plane *SP*. Clearly, with all coplanar entities, the classifier will have to decide whether they should appear in the part *Above* or the part *Below* in the final result. We view this as a *reclassification* problem: entities on *SP* must be reclassified either as lying above or below *SP* according to certain rules which will guarantee the correctness of the overall result.

Fortunately, such a set of rules is not difficult to come by. Consider first Figure 14.7 which depicts a two-dimensional projection of two cases where a face of the solid $S$ lies on the splitting plane *SP*. Obviously, each of the vertices of those faces will have one coplanar sector. In the case on the left, the coplanar face is expected to end up in the part *Below*, and on the right, the part *Above*. This can be achieved if the coplanar sectors and their edges are reclassified "to the side of the material."

We may also encounter coplanar edges that do not belong to any coplanar sectors, such as in the case of Figure 14.8. In the case depicted, the result of the sectioning cannot be represented as a 2-manifold, and we shall have to decide whether to treat the part *Above* as a single self-intersecting

```
void     splitgenerate(S, SP)
Solid    *S;
vector   SP;
{
     Edge          *e;
     HalfEdge      *he;
     Vertex        *v1, *v2;
     double        d1, d2, t, x, y, z;
     int           s1, s2;

     nvtx = 0;
     for(e = S->sedges; e != NIL; e = e->nexte)
     {
          v1 = e->he1->vtx;
          v2 = e->he2->vtx;
          s1 = comp((d1 = dist(v1->vcoord, SP)), 0.0, EPS);
          s2 = comp((d2 = dist(v2->vcoord, SP)), 0.0, EPS);
          if(s1 == -1 && s2 == 1 || s1 == 1 && s2 == -1)
          {
               t = d1 / (d1 - d2);
               x = v1->vcoord[0] + t*(v2->vcoord[0]-v1->vcoord[0]);
               y = v1->vcoord[1] + t*(v2->vcoord[1]-v1->vcoord[1]);
               z = v1->vcoord[2] + t*(v2->vcoord[2]-v1->vcoord[2]);
               lmev(e->he1, (he = e->he2->nxt), ++maxv, x, y, z);
               addsoov(he->prv->vtx);
          }
          else
          {
               if(s1 == 0) addsoov(v1);
               if(s2 == 0) addsoov(v2);
          }
     }
}

void     addsoov(v)
Vertex   *v;
{
     int       i;
     for(i=0; i<nvtx; i++) if(soov[i] == v) return;
     soov[nvtx++] = v;
}
```

**Program 14.2** Reduction step.

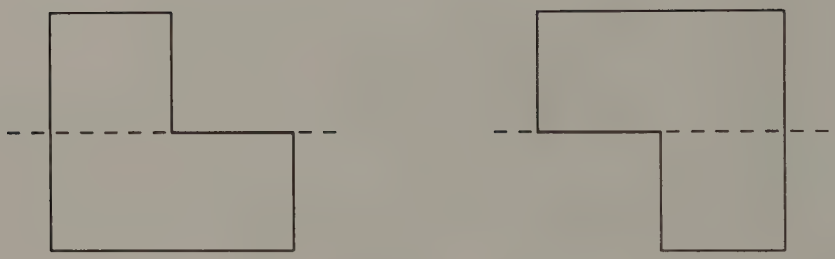


**Figure 14.7** Treatement of coplanar sectors.

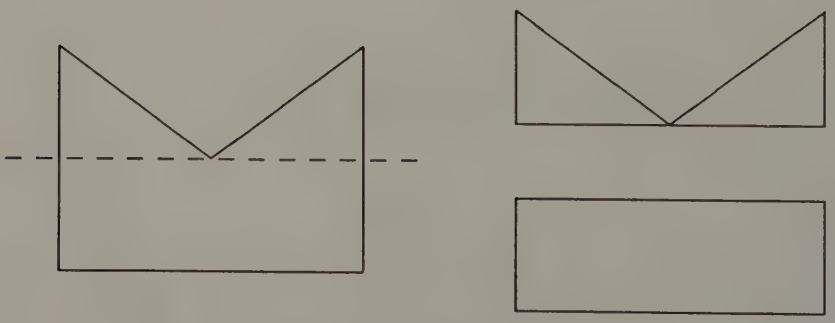


**Figure 14.8** Treatment of coplanar edges.

object or two touching wedges. We prefer the latter; this result is achieved if the coplanar edge is treated as lying below *SP*.

This leads us to the following outline for the vertex neighborhood classification:

1. Classify each edge of the neighborhood according to whether its final vertex lies above, on, or below *SP*. Tag the edge with the corresponding label ABOVE, ON, or BELOW.

2. Consider all labeled edges in their cyclic order around the vertex, and reclassify all edges tagged with ON by applying the following rules in the order given:

   (a) For each coplanar sector, reclassify the edges of the sector as lying "on the side of the material," i.e., if the face normal of the face of the sector is colinear with the normal of *SP*, the edges are relassified as BELOW, and otherwise ABOVE.

   (b) After applying the previous rule, ON-edges may appear in only four kinds of consecutive edge arrangements, namely

   ABOVE-ON-ABOVE
   ABOVE-ON-BELOW
   BELOW-ON-BELOW
   BELOW-ON-ABOVE.

   In these cases, the ON-edge is reclassified as BELOW, BELOW, ABOVE, and BELOW, respectively.

Observe that rule (b) is designed so that nonmanifold results will be represented as disconnected models.

### 14.5.2 Implementation of the Classifier

The overall flow of the classifier is given in Program 14.3. The neighborhood can be conveniently represented as terms of an array of structures that gives one halfedge of the sector and a classification code. Although not explicitly shown, we assume the array is global information accessible to all routines of this section.

The classifier simply loops through all coplanar vertices, and calls three procedures described in the sequel that implement the classification rules. The procedure `insertnulledges`, to be described in Section 14.6, examines the final classification, and inserts the appropriate null edges so as to divide the neighborhood.

```
# define ABOVE 1
# define BELOW -1
# define ON 0

struct
{
        HalfEdge        *sector;
        int             cl;
} nbr[MAXEDGESFORVERTEX];
int     nnbr;

void    splitclassify(SP)
vector  SP;
{
        int     i;

        nedg = 0;
        for(i=0; i<nvtx; i++)
        {
                getneighborhood(soov[i], SP);
                reclassifyonsectors(SP);
                reclassifyonedges();
                insertnulledges();
        }
}
```

**Program 14.3** Vertex neighborhood classifier.

```
void    getneighborhood(vtx, SP)
Vertex  *vtx;
vector  SP;
{
        HalfEdge        *he;
        vector          bisect;
        double          d;

        nnbr = 0;
        he = vtx->vedge;
        do
        {
                nbr[nnbr].sector = he;
                d = dist(he->nxt->vtx->vcoord, SP);
                nbr[nnbr++].cl = comp(d, 0.0, EPS);
                if(checkwideness(he) == 1)
                {
                        bisector(he, bisect);
                        nbr[nnbr].sector = he;
                        d = dist(bisect, SP);
                        nbr[nnbr++].cl = comp(d, 0.0, EPS);
                }
        }
        while((he = mate(he)->nxt) != vtx->vedge);
}
```

**Program 14.4** Initial classification.

### Initial Classification

Let us now elaborate the classification procedures. First, the procedure **getneighborhood** (Program 14.4) gathers the edges of the neighborhood into the array, and computes the initial classification for all edges by measuring the signed distances of the final vertices of each edge of the neighborhood.

For later parts of the algorithm it will be convenient to store "wide" sectors (i.e., sectors whose bounding edges make an angle not less than 180 degrees) as if they were replaced by two "small" ones. Therefore, the procedure constructs the bisector of each such sector, and inserts the sector also with the classification of the bisector. The procedures **checkwideness** and **bisector** for checking the wideness and constructing the bisector for a wide sector are left as exercises to the reader.

Observe that the literals ABOVE, ON, and BELOW were chosen in Program 14.3 so that the result of `comp` can be used directly as the classification code.

### Reclassification Procedures

The next two procedures implement the two reclassification rules stated.

The procedure `reclassifyonsectors` locates all sectors entirely coplanar with *SP*, and reclassifies them as explained above. In Program 14.5, the test for coplanarity is implemented by calculating the cross product of the face normal vector and the normal vector of *SP*, and testing whether the result is a null vector by means of a scalar product. If so, another scalar product tells us whether the normal vectors are actually colinear or opposite.

The final procedure `reclassifyonedges` is relatively straightforward; see Program 14.6. It just scans the neighborhood and applies the second reclassification rule.

## 14.6 INSERTION OF NULL EDGES

From the final result of the classification, it is easy to read where the vertex neighborhood is intersected by *SP*: if the classifications of two consecutive edges of the cycle differ, the sector in between has an intersection. What remains is the interpretation of this information, and the insertion of the appropriate null edges.

### 14.6.1 Insertion Rules

The goal of the insertion procedure is to separate all parts of the neighborhood that occur above *SP* from those that occur below it. To illustrate the insertion, let us study the sample case of Figure 14.9. On the left, the results of the classification are indicated. Wide sectors are also indicated. On the right, the expected configuration of null edges is shown. In this case, there are two distinct sequences of ABOVE-edges, and two null edges must be inserted to separate them. To represent the fact that a part of the leftmost wide sector will occur above *SP* even though its bounding edges are below *SP*, a "dangling" null edge must be inserted.

As we have the freedom of choosing an orientation to the null edges, we select to orient them always from the vertex below *SP* to the vertex above *SP*. This property will be very useful to us when joining the null edges to form section polygons.

```
void    reclassifyonsectors(SP)
vector  SP;
{
        Face            *f;
        vector          c;
        double          d;
        int             i;

        for(i=0; i<nnbr; i++)
        {
                f = nbr[i].sector->wloop->lface;
                cross(c, f->feq, SP);
                d = dot(c, c);
                if(comp(d, 0.0, EPS*EPS) == 0)
                {
                        d = dot(f->feq, SP);
                        if(comp(d, 0.0, EPS) == 1)
                        {
                                nbr[i].cl = BELOW;
                                nbr[(i+1)%nnbr].cl = BELOW;
                        }
                        else
                        {
                                nbr[i].cl = ABOVE;
                                nbr[(i+1)%nnbr].cl = ABOVE;
                        }
                }
        }
}
```

**Program 14.5** Reclassification of ON-sectors.

```
void    reclassifyonedges()
{
        int             i;

        for(i=0; i<nnbr; i++)
        {
                if(nbr[i].cl == ON)
                {
                        if(nbr[(nnbr+i-1)%nnbr].cl == BELOW)
                        {
                                if(nbr[(i+1)%nnbr].cl == BELOW)
                                        nbr[i].cl = ABOVE;
                                else    nbr[i].cl = BELOW;
                        }
                        else    nbr[i].cl = BELOW;
                }
        }
}
```

**Program 14.6** Reclassification of ON-edges.

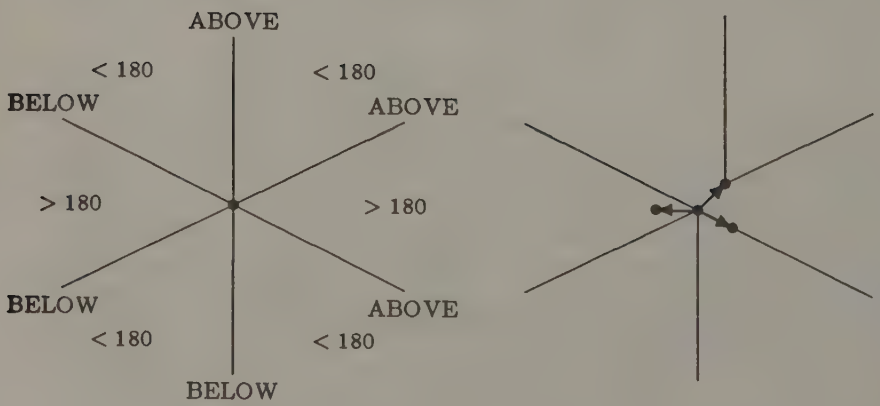



**Figure 14.9** Example of the insertion of null edges.

### 14.6.2 Insertion Procedure

The actual insertion procedure is greatly simplified by the convention of storing "wide" sectors twice in the array representation of the neighborhood. By virtue of this, wide sectors require no special treatment.

In the procedure `insertnulledges` of Program 14.7, the first `while`-loop locates the head of an ABOVE-sequence. If none is found, the algorithm terminates: all edges are on the same side of *SP*. Otherwise, the third `while`-loop locates the final sector of the sequence; thereafter, the null edge can be inserted. Finally, the last `while`-loop locates the start of the next sequence. The second, "perpetual" `while(1)`-loop will be exited when the head of the first sequence is hit again.

If wide sectors are present, the head and the tail of a sequence may in fact be the same sector; in this case, a "dangling" null edge will be inserted, as required. The reader is urged to examine the operation of the insertion algorithm for the sample case of Figure 14.9.

## 14.7 JOINING OF NULL EDGES

The joining step is supposed to combine the null edges (that mark individual intersection points) created in the previous step together to create null faces (that mark the section polygons). This is accomplished by connecting the end vertices of neighbor null edges with new edges.

### 14.7.1 Joining Order

To work properly, the joining algorithm must decide which null edges are joined. Fortunately, it can use the topological relationships available in the half-edge data structure to make this decision. Note that except for their geometry, null edges are quite ordinary edges of the half-edge data structure. In particular, they are linked to halfedges, and their neighboring loops and faces can be accessed.

This gives us one way to characterize "neighbor" null edges: they must occur in the same face. We can go further by observing the fact that null edges are always topologically oriented from the vertex below *SP* to the vertex above *SP*. This means that two null edges can be combined only by joining their head vertices with each other with a new edge, and respectively their tail vertices. This is equivalent to the requirement that halves of the null edges must occur in the same face, but in opposite orientations. These tests are implemented in Program 14.8.

Unfortunately, as a face may contain any number of null edges, topological neighborship does not alone determine the proper joining sequence

```
void    insertnulledges()
{
        int             start, i;
        HalfEdge        *head, *tail;

        i = 0;
        while(!(nbr[i].cl == BELOW && nbr[(i+1)%nnbr].cl == ABOVE))
                if(++i == nnbr) return;
        start = i;
        head = nbr[i].sector;
        while(1)
        {
                while(!(nbr[i].cl == ABOVE &&
                        nbr[(i+1)%nnbr].cl == BELOW))
                        i = (i+1) % nnbr;
                tail = nbr[i].sector;
                lmev(head, tail, ++maxv,
                        head->vtx->vcoord[0],
                        head->vtx->vcoord[1],
                        head->vtx->vcoord[2]);
                sone[nedg++] = head->prv->edg;
                while(!(nbr[i].cl == BELOW &&
                        nbr[(i+1)%nnbr].cl == ABOVE))
                {
                        i = (i+1) % nnbr;
                        if(i == start) return;
                }
        }
}
```

**Program 14.7** Insertion of null edges.

```
int             neighbor(h1, h2)
HalfEdge        *h1, *h2;
{
        return(h1->wloop->lface == h2->wloop->lface &&
               ((h1 == h1->edg->he1 && h2 == h2->edg->he2) ||
                (h1 == h1->edg->he2 && h2 == h2->edg->he1)));
}
```

**Program 14.8** Neighborship test.

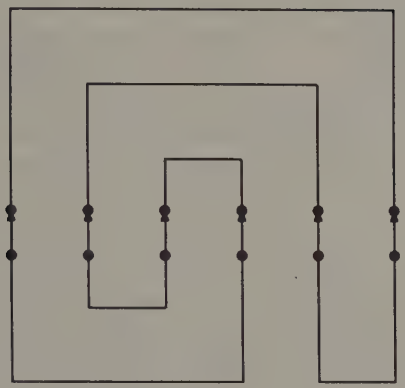


**Figure 14.10** Joining example.

uniquely. For a specific example, consider the case of Figure 14.10. The leftmost null edge can be joined topologically with three of the five other null edges marked. The figure also demonstrates that the correct neighbor is *not* necessarily either of the null edges found by scanning the halfedges of the face to the left and to the right of a null edge.

The obvious solution is to pick the *nearest* null edge to a given one for the joining operation. We implement this by considering null edges of *sone* in a lexicographically sorted order along $x$, $y$, and $z$. That is, null edges with at identical $x$ are ordered along $y$; if even the $y$-coordinates are equal, $z$ is finally used. Of course, equalities must be determined with comp. Because all null edges lie on $SP$, the lexicographic order together with the topological neighborship defines the correct joining sequence uniquely.

### 14.7.2 Implementation of the Joining Algorithm

The operation of the joining algorithm consists of processing the null edges in the lexicographic order, and determining whether the next edge can be combined with any yet unfinished section polygon(s). During the lexicographic sweep, not yet finished section polygons are represented in terms of two pointers to halfedges corresponding with the current loose ends of the section polygon.

The detailed operation of the joining algorithm is best explained with help of the Figure 14.11. In this sample case, the block (a) should be split into two smaller ones (b). After the vertex neighborhood classification, we have the arrangement of null edge shown in the plane model (c); dots

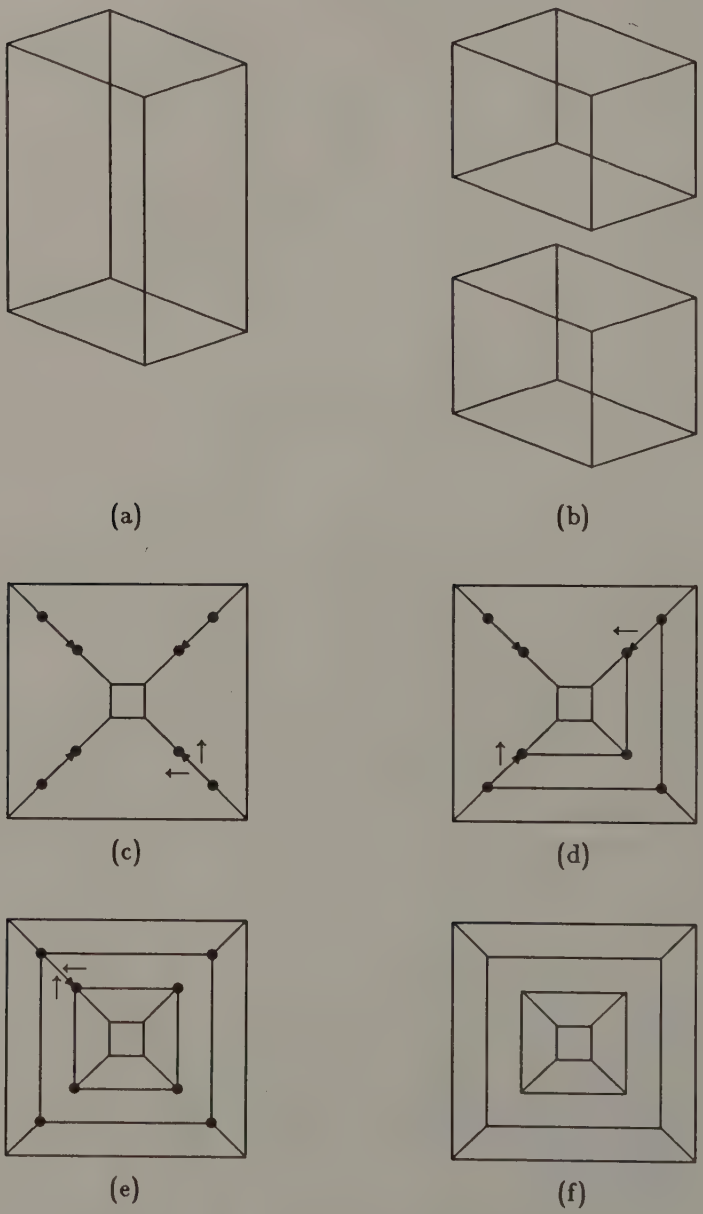



**Figure 14.11** Operation of the joining algorithm.

joined with a large arrow denote the null edges and their orientation.

The joining algorithm starts by picking a null edge (actually, the lexicographically smallest one); its two halves form the initial two loose ends of a section polygon. This is indicated with the smaller arrows. When new null edges are added to the section polygon, the ends are updated. Null edges can be removed when they do not form a loose end anymore (see Figure 14.11(d)). Finally, two loose ends may be combined if they can both by joined with a new null edge (e).

The procedure of Program 14.9 implements the joining algorithm. The algorithm begins by sorting *sone* lexicographically by $x$, $y$, and $z$ by a procedure **sortnulledges**. Thereafter, it retrieves each null edge in their sorted order by means of the procedure **getnextnulledge**, and performs the appropriate operations on them. The procedures **sortnulledges** and **getnextnulledge** are not elaborated.

In the general situation, some loose ends will already exist. By means of the procedure **canjoin**, we learn for both halves of the next edge whether the half can be joined with a loose end. If so, the actual joining is performed. After joining, the edge of the loose end can be cut, if the other half of the edge is not loose. The procedure **isloose** is assumed to determine whether its argument halfedge is a loose end.

The set of loose ends is maintained by the procedure **canjoin**. If a current loose end can be joined, it can be removed immediately from the set, because a joining operation will occur. Otherwise, the nonmatching half of the new null edge itself becomes a loose end.

The remaining pieces of the joining algorithm are **join**, for combining two null edges, and **cut**, for removing a null edge after it has been combined. These procedures can readily be implemented with Euler operators as in Program 14.10.

Procedure **join** basically has to determine whether the two null edges occur in the same loop in order to decide which Euler operators to use for combining null edges. Care must be taken to make sure that potential internal loops end up in correct faces after a MEF. The procedure **laringmv** fills this need. To make **join** directly useful for the purposes of the next chapter, the **lkef**'s are guarded by **if**-statements that check that the edges are not already connected with each other.

The completion of a section polygon is detected in procedure **cut**: when the same loop appears on both sides of a null edge to be removed, it must be the last one remaining in that section polygon (see Figure 14.11(e,f)), and **cut** can store a pointer to the face into *sonf*.

Figure 14.12 gives an example of the operation of the joining algorithm. The eight null edges are sorted here along $x$ and $z$; they all have the equal

```
static HalfEdge *ends[30];
static int      nend;

void    splitconnect()
{
        Edge            *nextedge, *getnextnulledge();
        HalfEdge        *h1, *h2, *canjoin();

        nfac = 0;
        sortnulledges();
        while(nextedge = getnextnulledge())
        {
                if(h1 = canjoin(nextedge->he1))
                {
                        join(h1, nextedge->he1);
                        if(!isloose(mate(h1))) cut(h1);
                }
                if(h2 = canjoin(nextedge->he2))
                {
                        join(h2, nextedge->he2);
                        if(!isloose(mate(h2))) cut(h2);
                }
                if(h1 && h2) cut(nextedge->he1);
        }
}
HalfEdge        *canjoin(he)
HalfEdge        *he;
{
        HalfEdge        *ret;
        int             i, j;

        for(i = 0; i<nend; i++)
                if(neighbor(he, ends[i]))
                {
                        ret = ends[i];
                        for(j=i+1; j<nend; j++)
                                ends[j-1] = ends[j];
                        nend--;
                        return(ret);
                }
        ends[nend++] = he;
        return((HalfEdge *) NIL);
}
```

**Program 14.9** The joining algorithm.

```
void            join(h1, h2)
HalfEdge        *h1, *h2;
{
        Face    *oldf, *newf;

        oldf = h1->wloop->lface;
        newf = (Face *) NIL;
        if(h1->wloop == h2->wloop)
        {
                if(h1->prv->prv != h2)
                        newf = lmef(h1, h2->nxt, ++maxf);
        }
        else    lmekr(h1, h2->nxt);
        if(h1->nxt->nxt != h2)
        {
                lmef(h2, h1->nxt, ++maxf);
                if(newf && oldf->floops->nextl)
                        laringmv(oldf, newf);
        }
}

void            cut(he)
HalfEdge        *he;
{
        if(he->edg->he1->wloop == he->edg->he2->wloop)
        {
                sonf[nfac++] = he->wloop->lface;
                lkemr(he->edg->he1, he->edg->he2);
        }
        else    lkef(he->edg->he1, he->edg->he2);
}
```

**Program 14.10** Joining and cutting of null edges.

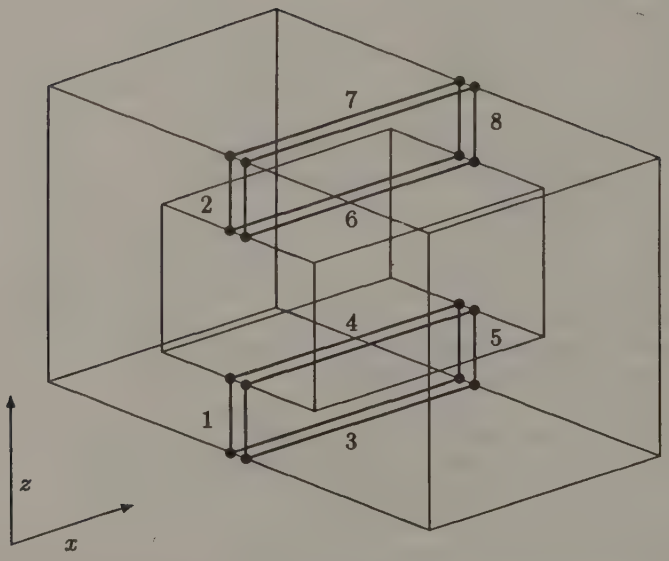



**Figure 14.12** Example of the joining algorithm.

$y$-coordinate. The numbers 1 ... 8 indicate the joining order of null edges. Note that in this case two section polygons are processed simultaneously.

## 14.8 CLASSIFICATION OF FACES

The final missing part of the splitting algorithm is the actual division of the sectioned solid into two solids representing the results. Fortunately, the joining algorithm supplies useful information for completing the remaining task.

Recall that the joining algorithm inserts a "null face" along the section polygon. The null face will contain two loops running through pairwise coincident vertices. Also, recall that null edges were generated so that they always point "up," i.e. so that their starting vertex belongs to the part below the splitting plane *SP*, and their final vertex similarly to the part above *SP*. By examining the joining algorithm (particularly, procedures `neighbor`, Program 14.8 and `join`, Program 14.10) one can see that "down" vertices are always connected with "down" vertices, and similarly "up" vertices with "up" vertices. Therefore, we know that the "inner" loop should appear in the part *Above*, and the "outer" loop in the part *Below*.

```
void      splitfinish(S, Above, Below)
Solid     *S, **Above, **Below;
{
     int        i;

     for(i=0; i<nfac; i++)
          sonf[nfac+i] = lmfkrh(sonf[i]->floops->nextl, ++maxf);
     *Above = (Solid *) new(NIL, SOLID);
     *Below = (Solid *) new(NIL, SOLID);
     classify(S, *Above, *Below);
     cleanup(*Above);
     cleanup(*Below);
}
```

**Program 14.11** Generation of results.

This makes the actual separation of the resulting solids straightforward (see Program 14.11). First, the null faces are transformed to two section faces by means of a MFKRH. This effectively increases the connectivity of the half-edge data structure so that it now models a disconnected solid.

The heart of the final phase is the classification of the faces of the now disconnected original solid into the solids *Above* and *Below*, the task of procedure `classify`. Observing that of all section faces, we already know where they should go, and that all other faces can be classified according to their topological neighborhood relationships to faces already classified, `classify` is easily implemented as a recursive depth-first-search algorithm that starts from the section faces and recursively moves all neighbors of a face that have not yet been moved from $S$ to *Above* or *Below*. The implementation of Program 14.12 uses procedures `addlist` and `dellist` described in Chapter 11 to manipulate the list of faces of a solid directly.

Note that `classify` only constructs the lists of faces of *Above* and *Below*—their lists of edges and vertices remain empty. The procedure `cleanup` is assumed to reconstruct these lists based on the list of faces. Its implementation is straightforward and is left to the reader.

## 14.9 SLICING

The splitting algorithm described above works on the half-edge data structure of a solid, and modifies it directly in order to create the results. Hence the original object disappears. A simple variation of the basic algorithm

```
void    classify(S, Above, Below)
Solid   *S, *Above, *Below;
{
        int             i;

        for(i = 0; i < nfac; i++)
        {
                movefac(sonf[i], Above);
                movefac(sonf[nfac+i], Below);
        }
}

void    movefac(f, s)
Face    *f;
Solid   *s;
{
        Loop            *l;
        HalfEdge        *he;
        Face            *f2;

        dellist(FACE, f, f->fsolid);
        addlist(FACE, f, s);
        l = f->floops;
        while(l)
        {
                he = l->ledg;
                do
                {
                        f2 = mate(he)->wloop->lface;
                        if(f2->fsolid != s)
                                movefac(f2, s);
                }
                while((he = he->nxt) != l->ledg);
                l = l->nextl;
        }
}
```

**Program 14.12** Classification of faces.

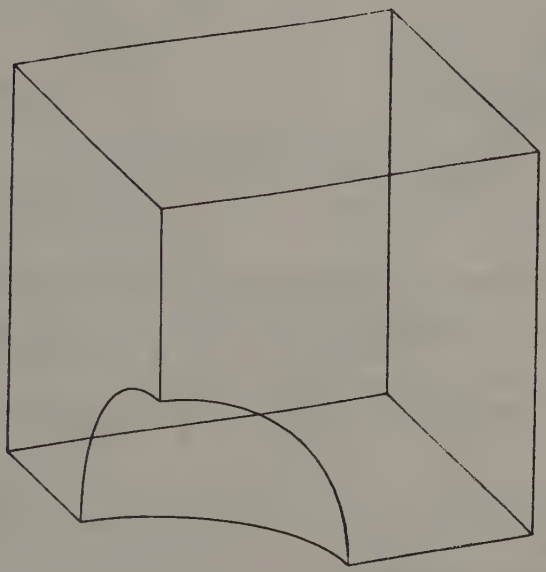


**Figure 14.13** A slicing operation.

generates the section polygons only and leaves the original object intact. We call this operation *slicing*.

Slices are useful for visualizing complicated objects. Sometimes they can be used also for modeling purposes, say, for the design of a container for a given part. For an example, see Figure 14.13.

To modify the splitting algorithm to a slicing algorithm, only modest changes in the joining algorithm and the later steps of the splitting algorithm are necessary. In particular, the procedure `join` that builds section polygons can be modified to build the polygons as a new solid *slice*, instead of manipulating the solid itself. To nullify the effects of the algorithm to the original solid, all new vertices created by it should be deleted.

## BIBLIOGRAPHIC NOTES

Figures 14.2, 14.3, 14.4, 14.5, and 14.6 have been previously published in the article "Boolean operations of 2-manifolds through vertex neighborhood classification" by M. Mäntylä in *Transactions on Graphics*, Volume 5, Number 1. ©1986, Association for Computing Machinery, Inc. Reprinted by permission.

## PROBLEMS

14.1. Implement the procedure

```
void      bisector(he, bisect)
HalfEdge  *he;
vector    bisect;
```

that constructs a vector along the bisector of the sector defined by *he*.

14.2. Implement the procedure

```
void    cleanup(s)
Solid   *s;
```

that reconstructs the lists of edges and vertices of its argument solid so that they will contain those and only those entities adjacent to the faces of the solid.

*Hint:* Read again last the paragraph of Section 10.4.1.

14.3. Outline the implementation of a slicing procedure based on the splitting algorithm.

# Chapter 15

# BOOLEAN SET OPERATIONS

An algorithm for evaluating Boolean set operations of solid objects is one of the most powerful facilities available in a solid modeler. In modelers based on boundary representations, the Boolean set operations algorithm is also the technically most demanding component. This chapter describes a Boolean set operations algorithm that can work for essentially all possible arrangements of polyhedral objects. As we shall see, the algorithm is similar in general structure to the splitting algorithm of the previous chapter.

## 15.1 INTRODUCTION

A Boolean set operations algorithm that can be used to unite, intersect, or subtract solid objects with each other is an essential component of any solid modeling system. For human users, set operations offer a tool for describing complex objects in terms of a series of operations on simpler components. For various algorithms, set operations offer a tool for describing physical processes applied to objects. For instance, an algorithm for verifying code for numerically controlled (NC) machine tools can use set operations to model effects in individual machining operations, or for determining collisions between the tool and the fixtures [58].

Unfortunately, Boolean set operations algorithms for boundary representations are in general plagued by two kinds of problems: First, to be effective, a set operations algorithm must be able to treat all possible kinds of geometric intersections that may appear between faces, edges, and vertices of the two objects. The proper treatment of all cases easily leads to a

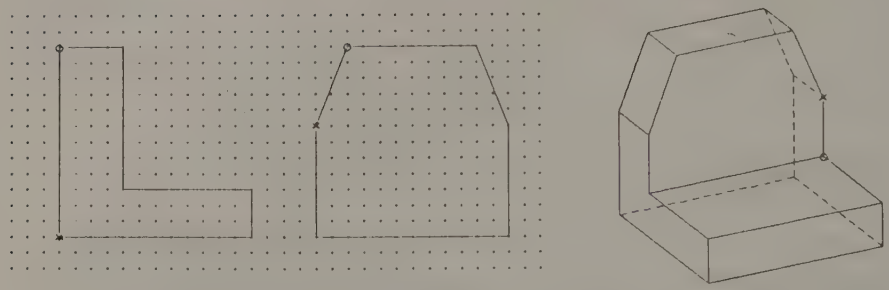


**Figure 15.1** Set operation of two extruded profiles.

very hairy case analysis. Second, the very case analysis must be based on various tests for overlap, coplanarity, and intersection which are difficult to implement robustly in the presence of numerical errors.

To appreciate the difficulty of the set operations, consider Figure 15.1 that depicts the set intersection of two extruded profiles. In the figure, the two top views depict two orthogonal profiles. The solid shown in the bottom right view is generated by sweeping the profiles appropriately to generate overlapping solids, and calculating their Boolean intersection. Clearly, the swept objects intersect each other in various "special" ways. In this case a set operations algorithm will have difficulties if it is not prepared to deal with coplanar overlapping faces such as the bottom face of the resulting object.

Observe that the code for overlapping coplanar faces must be able to handle robustly a two-dimensional set operation of two polygons, a task that even alone is quite challenging for a programmer uninitiated in geometry. All the program codes required to break the problem into solvable cases, and the codes for each case sum up to a voluminous and complicated algorithm whose correctness is difficult to establish.

For the purposes of GWB, we shall seek a different path to complete set operations. The algorithm to be presented breaks the Boolean set operations problem into a set of simpler problems of just two major kinds that can be handled with a relatively straightforward case analysis. As to be elaborated in the subsequent sections, the main underlying tasks are very similar in nature with the vertex neighborhood classification of the splitting algorithm described in the previous chapter.

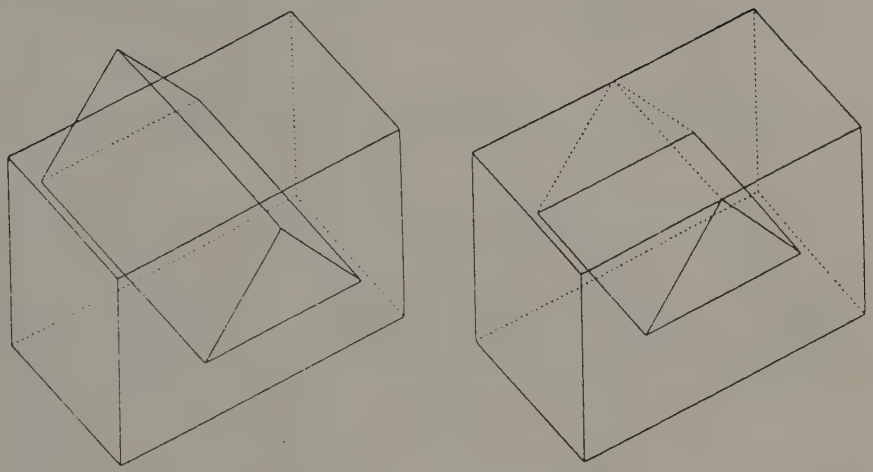


**Figure 15.2** Set operations are not closed for 2-manifolds.

## 15.2 STATEMENT OF THE PROBLEM

The input of the Boolean set operations algorithm will consist of two half-edge data structures, $A$ and $B$. Both $A$ and $B$ are assumed to be "correct" models of a three-dimensional solid object. In particular, they should be 2-manifolds, i.e., satisfy the following criteria:

1. Every edge belongs to exactly two flat faces.
2. Every vertex is surrounded by a single cycle of edges and faces.
3. Faces many not intersect each other except at common edges or vertices.

Our aim is to devise an algorithm that can calculate a new solid that represents the desired regularized Boolean combination of $A$ and $B$, i.e., $A \cup^* B$, $A \cap^* B$, or $A \setminus^* B$, where $\cup^*$, $\cap^*$, and $\setminus^*$ are the regularized counterparts of the ordinary (point) set operations union $\cup$, intersection $\cap$, and set difference $\setminus$.

Unfortunately, we cannot require that the resulting solid will always satisfy both of the requirements 1 and 3 above. This is so because 2-manifolds are not closed under set operations, i.e., a Boolean combination of two 2-manifolds is not necessarily a 2-manifold. For instance, the set difference of the block and the wedge of Figure 15.2 has four faces meeting at an edge, and criterion 1 is not satisfied.

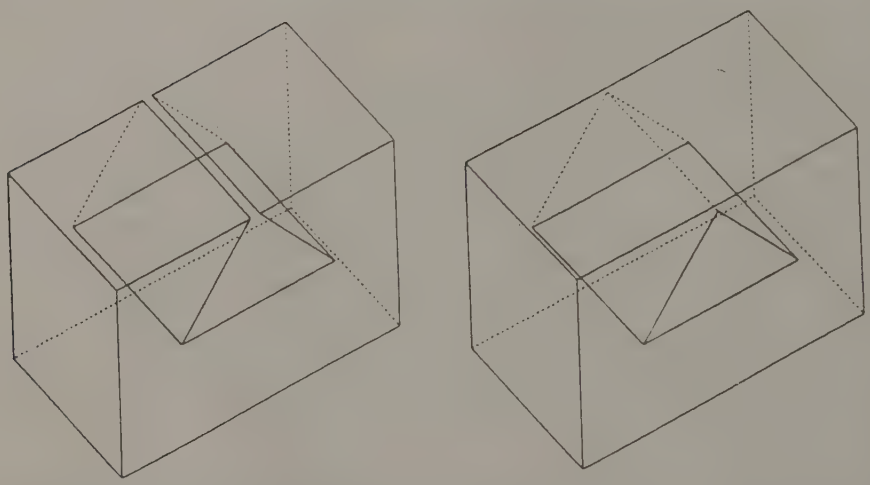


**Figure 15.3** Pseudomanifold representations of a nonmanifold.

We accept this limitation because a winged-edge data structure cannot represent objects such as the result of Figure 15.2 directly. The only resort available is to represent the nonmanifold as a "pseudomanifold," i.e., as if it was one or the other of the objects depicted in Figure 15.3. In particular, in the case on the left the wedge extends outside the top face of the block, while on the right the wedge only intersects the front and the back faces of the block. In the first case, the object has two entirely coincident edges, while in the second case it has an edge lying on a face.

Hence, we shall require that the result of the set operations algorithm satisfies all other criteria except 3. Instead of 3, a more permissive criterion will be required:

3′. Faces of the polyhedron may not intersect each other at their internal points. The only other kinds of intersections are the following:

- Some edges of the polyhedron are entirely coincident. In this case, the edge neighborhoods of these edges must be distinct.
- Some edges may lie on a face. In this case, both faces adjacent to the edge must lie on the same side of the face.
- Some vertices may lie on a face. In this case, all edges and faces adjacent to the vertex must lie on the same side of the face.
- Some vertices may lie on an edge or on a vertex. In these cases, all edges and faces adjacent to the vertex must lie on the same

side of the surface defined by the faces of the incident edge or vertex.

That is, the surface of the resulting object may "touch" itself at some edges or vertices, but not intersect itself properly.

## 15.3 BOUNDARY CLASSIFICATION

The splitting algorithm of Chapter 14 is based on reducing the global problem into a collection of local vertex neighborhood classification problems, and processing each vertex neighborhood according to rules that guarantee the correctness of the result. The set operations algorithm and its classifier will follow the same general approach.

To see the close relationship between the splitting problem and the set operations, recall that the splitting algorithm divides a solid $S$ in two sets (*Above*, *Below*) according to a splitting plane $SP$. For set operations we shall need the more general splitting of a polyhedron $A$ according to a reference polyhedron $B$. More rigorously, let us denote the bounding surface of $A$ by $b(A)$. Then the two objects resulting from the general splitting operation will be denoted by $AinB$ (for the part of $b(A)$ inside $B$) and $AoutB$ (for the part of $b(A)$ outside $B$).[1]

Given two solids $A$ and $B$, the collection of four objects $AinB$, $AoutB$, $BinA$, and $BoutA$ formed by splitting $A$ against $B$ and symmetrically $B$ against $A$ are here called the *boundary classification* of $A$ and $B$. For instance, Figure 15.4(a) depicts two intersecting bricks; their intersection curve is indicated with heavy lines. The resulting boundary classification is shown in (b). (Observe that each component of the classification can be represented as a solid in its own right that generally has some nonplanar "intersection" faces.)

From the boundary classification of $A$ and $B$, all their Boolean combinations are readily computed:

$$\begin{aligned} A \cup B &= AoutB \oplus BoutA \\ A \cap B &= AinB \oplus BinA \\ A \setminus B &= AoutB \oplus (BinA)^{-1} \end{aligned} \tag{15.1}$$

where $\oplus$ denotes the gluing operation of Chapter 12, and $(BinA)^{-1}$ denotes the "complement" of $BinA$, i.e., $BinA$ with the orientation of all faces reversed.

[1] Note that this is exactly what the splitting algorithm does if we think of the halfspace "above" $SP$ as a polyhedron $B$ bounded by just one infinite face.

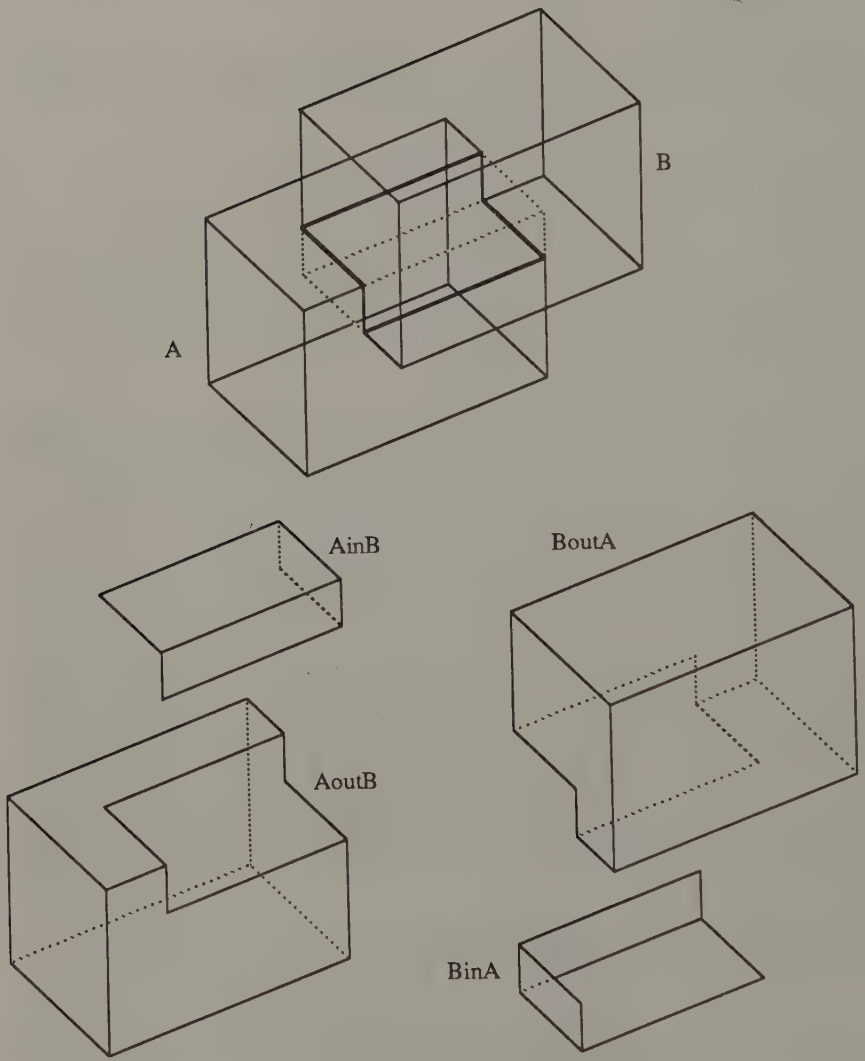



**Figure 15.4** Boundary classification of two bricks.

In the splitting algorithm, the vertex neighborhood classifier treats edges and faces lying on the splitting plane by reclassifying them as lying above or below the plane. The reclassification rules are designed so as to guarantee the regularity of the result. A similar approach can be followed also in the classifier for set operations. However, in this case the reclassification rules will be somewhat more complicated.

To provide insight into the reclassification rules, it is useful to consider the *8-way boundary classification* of $A$ and $B$ that adds the components $AonB^+$, $AonB^-$, $BonA^+$, and $BonA^-$ to the components of the 4-way classification ($AinB$, $AoutB$, $BinA$, $BoutA$). The component $AonB^+$ consists of those parts of $b(A)$ that lie on $b(B)$ so that the face normals of the respective faces are equal, while $AonB^-$ consists of the overlapping parts where the normals are opposite. The components $BonA^+$ and $BonA^-$ are defined analogously.

Figure 15.5 illustrates the 8-way boundary classification of two objects $A$ and $B$ shown in three views. For clarity, the "in"- and "on"-components of the classification are shaded.

From the components of the 8-way classification, the result of a set operation can be computed as follows:

$$\begin{aligned} A \cup B &= AoutB \otimes BoutA \otimes AonB^+ \\ A \cap B &= AinB \otimes BinA \otimes AonB^+ \\ A \setminus B &= AoutB \otimes (BinA)^{-1} \otimes AonB^- \end{aligned} \tag{15.2}$$

(The reader is encouraged to check that these equations indeed give the correct regularized results for the objects of Figure 15.5.)

It would be perfectly possible to construct a set operations algorithm that computes the full 8-way boundary classification, and works out the result according to Equations 15.2. Because of the inherent burden of this computation, we do not follow this approach directly. Instead, our algorithm calculates a special 4-way boundary classification where the "on"-components of the 8-way classification are lumped with the components of the 4-way classification so as to make Equations 15.1 and 15.2 equivalent.

A set of reclassification rules with this property can be written as follows:

| $Op$ | $AonB^+$ | $AonB^-$ | $BonA^+$ | $BonA^-$ |
|---|---|---|---|---|
| $\cup$ | $AoutB$ | $AinB$ | $BinA$ | $BinA$ |
| $\cap$ | $AinB$ | $AoutB$ | $BoutA$ | $BoutA$ |
| $\setminus$ | $AinB$ | $AoutB$ | $BoutA$ | $BoutA$ |

(15.3)

For instance, the first row indicates that for set union, $AonB^+$ should be treated as being a part of $AoutB$, and $AonB^-$ as a part of $AinB$. $BonA^+$

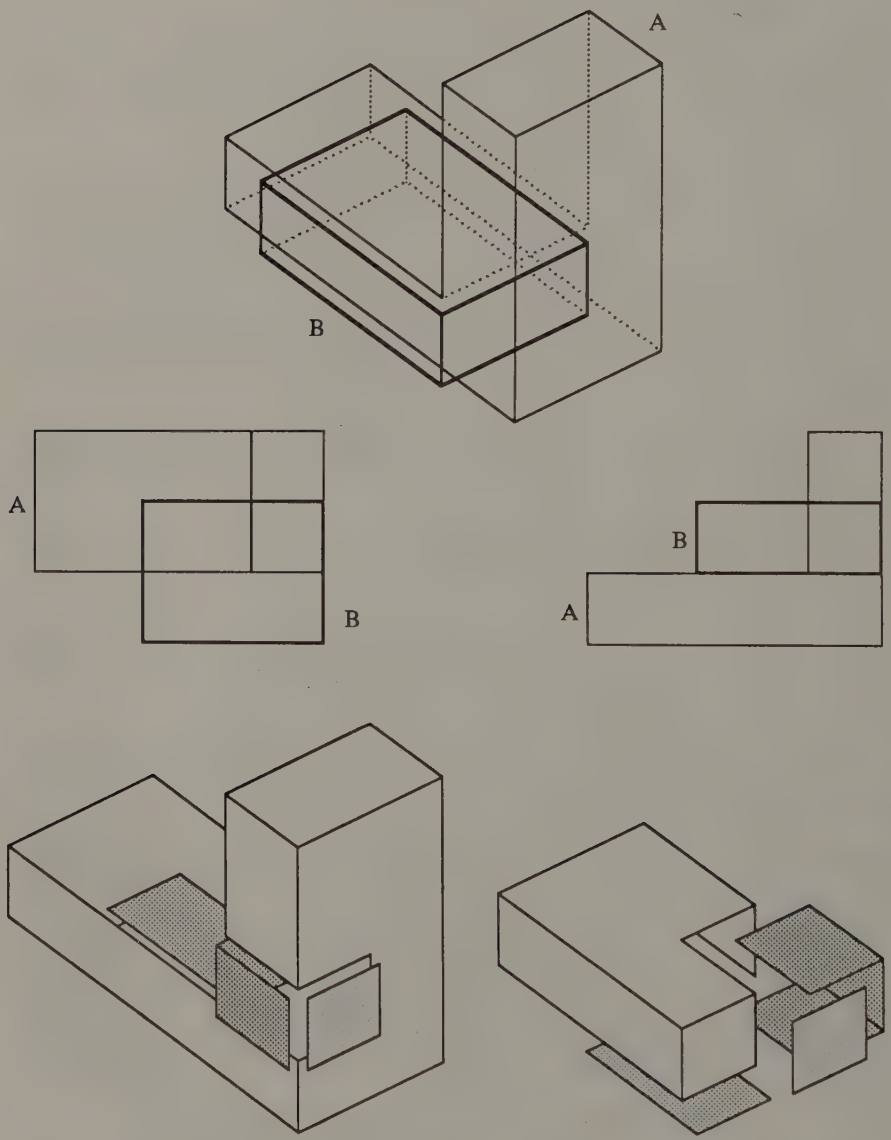



**Figure 15.5** 8-way boundary classification of two polyhedra.

and $BonA^-$ are both treated as a part of $BinA$. Hence, only $AonB^+$ will appear in the final result.

It should be stressed that these rules will be implemented directly in the vertex neighborhood classifier routines, and that the higher levels of the algorithm can effectively ignore all "on"-cases. (Observe the analogy of the rules of Equations 15.3 and the reclassification of "on"-edges in the splitting algorithm of Chapter 14.)

## 15.4 THE ALGORITHM

Let us now start the description of the set operations algorithm. The implementation of Program 15.1 is written in a style stressing the close relationship of splitting and set operations; the reader is urged to compare it with the splitting algorithm of Program 14.1.

For use of the vertex neighborhood classifier, the algorithm uses three arrays *sonvv*, *sonva*, and *sonvb* for representing the set of conincident vertices of $A$ and $B$, the set of vertices of $A$ on a face of $B$, and the set of vertices of $B$ on a face of $A$, respectively. Similarly to the splitting algorithm, the set operations algorithm will also need arrays for representing the sets of null edges (*sonea*, *soneb*) to be generated on $A$ and $B$, and the sets of null faces (*sonfa*, *sonfb*). All these arrays are assumed to be global data available to all parts of the algorithm.

The algorithm takes four arguments. $A$ and $B$ point at the solids to be combined by means of the set operation *op* represented by means of literal values `UNION`, `INTERSECT`, and `DIFFERENCE`. The argument *res* will be filled with a pointer to the resulting solid.

The algorithm first evaluates all face equations and arranges the global variables *maxv* and *maxf* so that they can be used to generate globally unique new identifiers for vertices and faces. The procedure `updmaxnames` is assumed to increment the face and vertex identifiers of its argument solid so that they do not overlap the ranges 1–*maxf* and 1–*maxv*, and update *maxv* and *maxf* appropriately.

The procedures `setopgenerate`, `setopclassify`, `setopconnect`, and `setopfinish` have similar roles as their counterparts in Program 14.1. They will be elaborated in the following sections.

## 15.5 REDUCTION STEP

First, let us consider the procedure `setopgenerate` responsible for reducing the set operations problem into a collection of vertex neighborhood classification problems. This task can be formulated as follows:

```
struct
{
        Vertex  *va, *vb;
}       sonvv[MAXONVERTICES];
int     nvtx;
struct
{
        Vertex *v;
        Face   *f;
}       sonva[MAXONVERTICES], sonvb[MAXONVERTICES];
int     nvtxa, nvtxb;
Edge    *sonea[MAXNULLEDGES], *soneb[MAXNULLEDGES];
int     nedga, nedgb;
Face    *sonfa[MAXNULLFACES*2], *sonfb[MAXNULLFACES*2];
int     nfaca, nfacb;

int     setop(A, B, res, op)
Solid   *A, *B, **res;
int     op;
{
        Face    *f;

        for(f = A->sfaces; f != NIL; f = f->nextf)
                faceeq(f->flout, f->feq);
        for(f = B->sfaces; f != NIL; f = f->nextf)
                faceeq(f->flout, f->feq);
        getmaxnames(A);
        updmaxnames(B);
        setopgenerate(A, B);
        setopclassify(op);
        if(nedga == 0)
        {
                printf("setop: no intersections found\n");
                return(0);
        }
        setopconnect();
        setopfinish(A, B, res, op);
        return(1);
}
```

**Program 15.1** Outline of the set operations algorithm.

1. Locate all pairs of edges $e_A$ of $A$ and $e_B$ of $B$ that intersect each other properly, i.e., at an internal point of both edges. Subdivide both edges at their intersection point, i.e., replace each edge by two edges and a new vertex lying at the intersection point.

2. Locate all edges of $A$ that pass through a vertex of $B$. Subdivide all such edges at the intersection point.

3. Do step 2 symmetrically for edges of $B$ and vertices of $A$.

4. Locate all coincident pairs of vertices $v_A$ of $A$ and $v_B$ of $B$, and store them for later processing. (Of course, the resulting set includes at least all vertices added during steps 1 through 3.)

5. Locate all edges $e_A$ of $A$ that intersect a face $f_B$ of $B$ properly, i.e., at an internal point of $f_B$. Subdivide all such edges at the intersection point.

6. Do step 5 symmetrically for edges of $B$ and faces of $A$.

7. Locate all vertices $v_A$ of A that lie within a face $f_B$ of B and store the pair $(v_A, f_B)$ for later processing. (This set will include all vertices added during step 5.)

8. Locate all vertices $v_B$ of B that lie within a face $f_A$ of A and store the pair $(v_B, f_A)$ for later processing. (This set includes all vertices added in step 6.)

The reader is urged to compare these steps with those of the splitting algorithm of Chapter 14. In particular, note the analogy of steps 5–8 of this algorithm and the splitting algorithm.

To implement the steps 1–8 as a real procedure it is very helpful to make the further assumption that all faces of $A$ and $B$ are "maximal," i.e., that all coplanar neighbor faces have been combined, and all "inessential" edges that occur within a single face have been removed. Under this assumption, the search of intersecting entities can be organized as a comparison between all edges of one solid against the faces of the other solid and vice versa, and applying the following case analysis for each edge $e$ and face $f$:

1. The vertices of $e$ are on the same side of the plane of $f$. In this case, $e$ does not intersect $f$.

2. The vertices of $e$ are on different sides of the plane of $f$. That is, $e$ intersects the plane "properly," In this case, an intersection point can be calculated and classified against $f$ by means of the vertex-in-face procedure `contfv` of Chapter 13. This gives a second-level case

```
void      setopgenerate(A, B)
Solid     *A, *B;
{
     Edge      *e;

     nvtx = nvtxa = nvtxb = 0;
     for(e = A->sedges; e != NIL; e = e->nexte)
          processedge(e, B, 0);
     for(e = B->sedges; e != NIL; e = e->nexte)
          processedge(e, A, 1);
}

void      processedge(e, s, BvsA)
Edge      *e;
Solid     *s;
int       BvsA;
{
     Face      *f;

     for(f = s->sfaces; f != NIL; f = f->nextf)
          dosetopgenerate(e, f, BvsA);
}
```

**Program 15.2** Generation of relevant vertices.

analysis according to whether the intersection lies within the face, at an edge of the face, at a vertex, or without the face.

3. One or both of the vertices of $e$ lie on the plane of $f$. Similar to the case 2 above, they can be classified with `contfv`.

Except for the vertices, the case of $e$ being coplanar with $f$ can be ignored: all relevant intersections between $e$ and the edges of $f$ will be located when comparing $e$ against the (noncoplanar) neighbor faces of $f$.

The edge-face comparison is realized in Program 15.2 ultimately by calling the procedure `dosetopgenerate` of Program 15.3 that implements the above case analysis. The procedure first computes the signed distances of the endpoints of the query edge $e$ from the plane of the face $f$. If the signs of the distances differ, the edge must intersect the plane, and the intersection point $p = (x\ y\ z)$ is computed.

This leads us to the second level of the case analysis. First, the inclusion of $p$ within $f$ is examined by means of the procedure `contfp`, a variation

```
extern HalfEdge *hithe;
extern Vertex *hitvertex;

void      dosetopgenerate(e, f, BvsA)
Edge      *e;
Face      *f;
int       BvsA;
{
     Vertex        *v1, *v2;
     double        d1, d2, t, x, y, z;
     int           s1, s2, cont;

     v1 = e->he1->vtx;
     v2 = e->he2->vtx;
     s1 = comp((d1 = dist(v1->vcoord, f->feq)), 0.0, EPS);
     s2 = comp((d2 = dist(v2->vcoord, f->feq)), 0.0, EPS);
     if(s1 == -1 && s2 == 1 || s1 == 1 && s2 == -1)
     {
          t = d1 / (d1 - d2);
          x = v1->vcoord[0] + t * (v2->vcoord[0] - v1->vcoord[0]);
          y = v1->vcoord[1] + t * (v2->vcoord[1] - v1->vcoord[1]);
          z = v1->vcoord[2] + t * (v2->vcoord[2] - v1->vcoord[2]);
          if((cont = contfp(f, x, y, z)) > 0)
          {
               lmev(e->he1, e->he2->nxt, ++maxv, x, y, z);
               if(cont == 1)
                    addsovf(e->he1->vtx, f, BvsA);
               else if(cont == 2)
               {
                    lmev(hithe, mate(hithe)->nxt, ++maxv, x, y, z);
                    addsovv(e->he1->vtx, hithe->vtx, BvsA);
               }
               else if(cont == 3)
                    addsovv(e->he1->vtx, hitvertex, BvsA);
               processedge(e->he1->prv->edg, f->fsolid, BvsA);
          }
     }
     else
     {
          if(s1 == 0) dovertexonface(v1, f, BvsA);
          if(s2 == 0) dovertexonface(v2, f, BvsA);
     }
}
```

**Program 15.3** Edge-face comparison.

```
void      dovertexonface(v, f, BvsA)
Vertex    *v;
Face      *f;
int       BvsA;
{
     int           cont;

     if((cont = contfv(f, v)) == 1)
          addsovf(v, f, BvsA);
     else if(cont == 2)
     {
          lmev(hithe, mate(hithe)->nxt, ++maxv,
               v->vcoord[0], v->vcoord[1], v->vcoord[2]);
          addsovv(v, hithe->vtx, BvsA);
     }
     else if(cont == 3)
          addsovv(v, hitvertex, BvsA);
}
```

**Program 15.4** Vertex-face comparison.

of the procedure **contfv** that tests for the inclusion of a point $(x\ y\ z)$ in a face $f$. If $p$ intersects $f$ is some fashion (i.e., **contfp** $> 0$), $e$ is subdivided at $p$ by creating a new vertex $v$ on it. If **contfp** $= 1$, $p$ is known to lie properly within $f$, and the pair $v, f$ is inserted in the proper set of coplanar vertex-face pairs by means of the procedure **addsovf**.

If **contfp** $= 2$, $p$ lies on an edge $e'$ of $f$ available in the external variable *hithe* (see Program 13.4, page 223). In this case, also $e'$ is subdivided, and the pair of new vertices is inserted to the set of coincident vertices by means of the procedure **addsovv**. Finally, if $p$ lies at a vertex $v'$ of $f$ (available in the external variable *hitvertex*), the pair $v, v'$ is inserted.

If one or other of the distances of the endpoints of the query edge is (approximately) zero, a coplanar vertex has been located. They are handled by the procedure **dovertexonf** of Program 15.4 which is similar in structure to the Program 15.3, and is presented without further explanations.

Observe that after subdividing the query edge, **dosetopgenerate** will recursively call itself through the procedure **processedge** of Program 15.3 to make sure that all intersections of the subdivided edge with the solid are processed. Note also that the procedures **addsovf** and **addsovv** should check against duplicate inputs, because the objects to be stored will be encountered several times.

```
void    setopclassify(op)
int     op;
{
        int     i;

        nedga = nedgb = 0;
        for(i=0; i<nvtxa; i++)
                vtxfacclassify(sonva[i].v, sonva[i].f, op, 0);
        for(i=0; i<nvtxb; i++)
                vtxfacclassify(sonvb[i].v, sonvb[i].f, op, 1);
        for(i=0; i<nvtx; i++)
                vtxvtxclassify(sonvv[i].va, sonvv[i].vb, op);
}
```

**Program 15.5** Vertex neighborhood classification.

## 15.6 VERTEX NEIGHBORHOOD CLASSIFIER

After the reduction step, the set operations problem is reduced into a collection of vertex neighborhood classification problems of two kinds:

1. *Vertex-face classification*: The processing of a vertex of one solid that lies on a face of the other solid.

2. *Vertex-vertex classification*: The processing of a pair of coincident vertices of $A$ and $B$.

In strict analogy to the splitting algorithm, both of these problems are solved by a neighborhood classifier that inserts the proper null edges so as to split the vertex neighborhoods into subcycles inside and outside of the respective other polyhedron.

### 15.6.1 Vertex-Face Classification

In Program 15.5, the procedure `vtxfacclassify` serves the role of the vertex-face classifier. This procedure is essentially identical to the corresponding procedure for the splitting algorithm (Programs 14.3–7). The differences are the following:

1. Instead of the splitting plane $SP$, the plane of the face is used, and the notions of "above" and "below" are replaced by "in" and "out."

2. "On"-sectors are reclassified according to Equations 15.3 instead of the rules of Chapter 14.

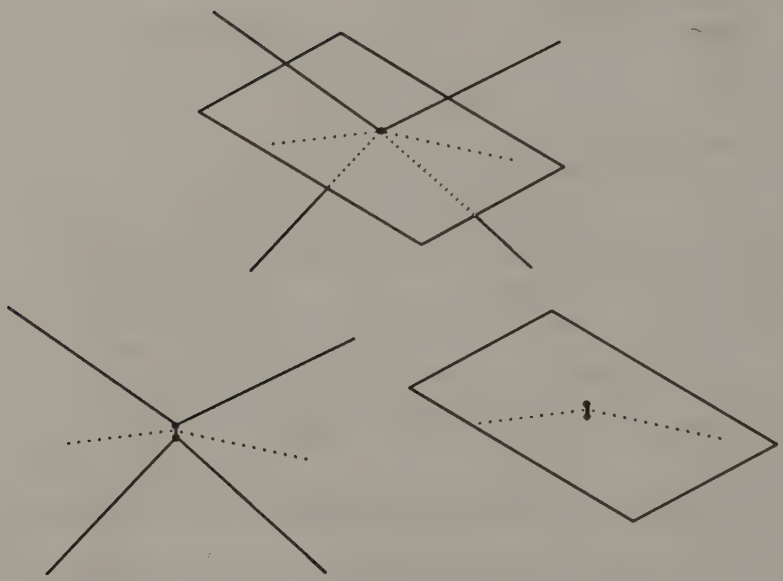


**Figure 15.6** Vertex-face classification.

3. In addition to splitting the vertex neighborhood, a reference to the intersection point must also be stored within the face so that the intersection polygon(s) can be drawn through the point.

As for the last item, the algorithm for joining null edges can work properly if the intersections are marked by inserting null edges as an internal loop into the face intersected; see Figure 15.6.

For space reasons, we shall not elaborate the vertex-face classifier any further.

### 15.6.2 Vertex-Vertex Classification

The vertex-vertex classifier is expected to compare two vertex neighborhoods (with coincident base vertices), determine their intersection, and insert the appropriate null edges so that the intersection polygons can be correctly generated. As this component forms the core of the algorithm, we shall present it in considerable detail.

An outline of the vertex-vertex classifier is shown in Program 15.6; the reader is urged to compare it with the analogous Program 14.3. The

```
# define OUT -1
# define IN 1
# define ON 0

struct nb
{
        HalfEdge        *he;
        vector          ref1;
        vector          ref2;
        vector          ref12;
} nba[MAXSECTORS], nbb[MAXSECTORS];
int     nnba, nnbb;

struct
{
        int             secta, sectb;
        int             s1a, s2a;
        int             s1b, s2b;
        int             intersect;
}       sectors[MAXSECTORS];
int     nsectors;

void    vtxvtxclassify(va, vb, op)
Vertex  *va, *vb;
int     op;
{
        setopgetneighborhood(va, vb);
        sreclsectors(op);
        srecledges(op);
        sinsertnulledges();
}
```

**Program 15.6** Vertex-vertex neighborhood classifier.

algorithm uses three arrays *nba*, *nbb*, and *sectors* to represent the sectors of the neighborhood of the two vertices and the intersections between sectors; these arrays are assumed to be accessible to all algorithms of this section.

### Search for intersecting sectors

Recall how the classifier for the splitting problem could work by locating all sectors of a neighborhood that intersect the splitting plane, and dividing the neighborhood along the intersecting sectors. Analogously, the vertex-vertex classifier of Program 15.7 works by considering pairs of sectors from the two neighborhoods, and testing for their intersection.

First, the algorithm will precompute some pieces of data of the two neighborhoods, and store them into the arrays *nba* and *nbb* for later use. Similarly to the splitting algorithm, it is very helpful to subdivide "wide" sectors whose bounding vectors make an angle larger than 180 degrees. These tasks are implemented by the Program 15.8

The main loop of the Program 15.7 compares two preprocessed sectors by means of the procedure `sectortest` described in the next subsection. If the two sectors are deemed to intersect, the classifications (`IN`, `ON`, `OUT`) of the bounding vertices of each sector with respect to the other sector are computed and stored into the array *sectors* for later processing.

### Sector intersection test

The sector intersection test is illustrated in Figure 15.7. A vector *int* along the intersection line can be calculated as the cross product of the plane normal vectors *n1* and *n2*:

$$int = n1 \times n2 \tag{15.4}$$

If the cross product vanishes, the sectors are coplanar and considered to intersect if they overlap. Otherwise, an inclusion test for *int* and the sectors is performed; if *int* occurs within both sectors, they intersect.

As shown in Figure 15.8, the inclusion is determined by examining the following cross products of *ref1*, *ref2*, and *int*:

$$\begin{aligned} ref &= ref1 \times ref2 \\ test1 &= ref1 \times int \\ test2 &= int \times ref2 \end{aligned} \tag{15.5}$$

If all three vectors are collinear, *int* is within the sector bounded by *ref1* and *ref2*.

```
void      setopgetneighborhood(va, vb)
Vertex    *va, *vb;
{
     HalfEdge   *ha, *hb;
     double     d1, d2, d3, d4;
     int        na, nb, i, j;

     nnba = nbrpreproc(va, nba);
     nnbb = nbrpreproc(vb, nbb);
     nsectors = 0;
     for(i=0; i<nnba; i++)
     {
          for(j=0; j<nnbb; j++)
          {
               if(sectortest(i, j))
               {
                    sectors[nsectors].secta = i;
                    sectors[nsectors].sectb = j;
                    d1 = dot(nbb[j].he->wloop->lface->feq,
                             nba[i].ref1);
                    d2 = dot(nbb[j].he->wloop->lface->feq,
                             nba[i].ref2);
                    d3 = dot(nba[i].he->wloop->lface->feq,
                             nbb[j].ref1);
                    d4 = dot(nba[i].he->wloop->lface->feq,
                             nbb[j].ref2);
                    sectors[nsectors].s1a = comp(d1, 0.0, EPS);
                    sectors[nsectors].s2a = comp(d2, 0.0, EPS);
                    sectors[nsectors].s1b = comp(d3, 0.0, EPS);
                    sectors[nsectors].s2b = comp(d4, 0.0, EPS);
                    sectors[nsectors++].intersect = 1;
                }
           }
     }
}
```

**Program 15.7** Search for intersecting sectors.

```
int             nbrpreproc(v, n)
Vertex          *v;
struct nb       n[];
{
    vector      bisec;
    HalfEdge    *he;
    int         i;

    i = 0;
    he = v->vedge;
    do
    {
        n[i].he = he;
        vecminus(n[i].ref1, he->prv->vtx->vcoord,
                 he->vtx->vcoord);
        vecminus(n[i].ref2, he->nxt->vtx->vcoord,
                 he->vtx->vcoord);
        cross(n[i].ref12, n[i].ref1, n[i].ref2);
        if(vecnull(n[i].ref12, EPS) ||
          (dot(n[i].ref12, he->wloop->lface->feq) > 0.0))
        {
            if(vecnull(n[i].ref12, EPS))
                inside(he, bisec);
            else
            {
                vecplus(bisec, n[i].ref1, n[i].ref2);
                bisec[0] = -bisec[0];
                bisec[1] = -bisec[1];
                bisec[2] = -bisec[2];
            }
            n[i+1].he = he;
            veccopy(n[i+1].ref2, n[i].ref2);
            veccopy(n[i+1].ref1, bisec);
            veccopy(n[i].ref2, bisec);
            cross(n[i].ref12, n[i].ref1, n[i].ref2);
            cross(n[i+1].ref12, n[i+1].ref1, n[i+1].ref2);
            i++;
        }
        i++;
    }
    while((he = mate(he)->nxt) != v->vedge);
    return(i);
}
```

**Program 15.8** Preprocessing of sectors.

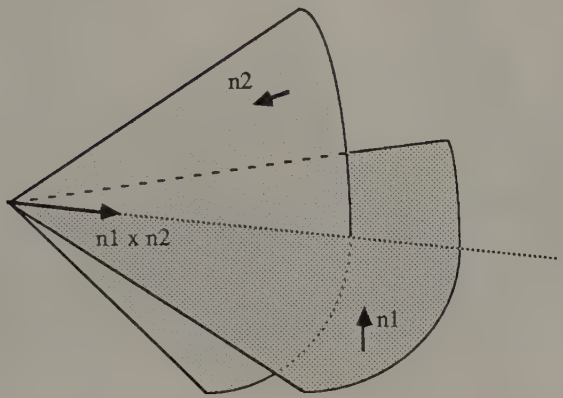



**Figure 15.7** Sector intersection test.

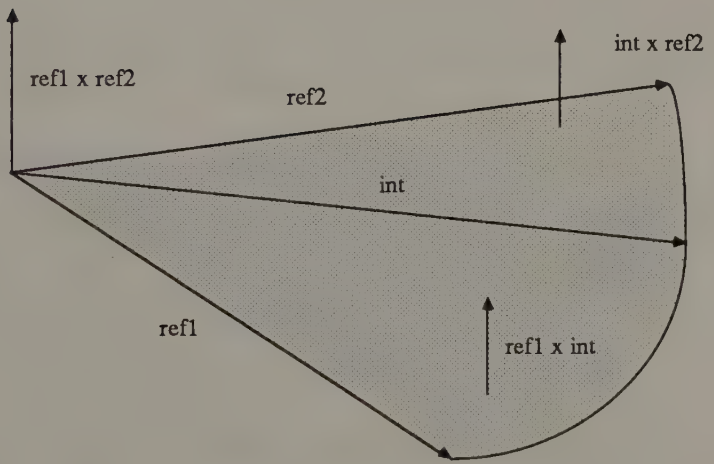



**Figure 15.8** Within-sector test.

```
int  sectortest(i, j)
int  i, j;
{
     vector    intrs;
     HalfEdge  *h1, *h2;
     int       c1, c2;

     h1 = nba[i].he;
     h2 = nbb[j].he;
     cross(intrs, h1->wloop->lface->feq, h2->wloop->lface->feq);
     if(vecnull(intrs, EPS)) return(sectoroverlap(h1, h2));
     c1 = sctrwithin(intrs, nba[i].ref1, nba[i].ref2, nba[i].ref12);
     c2 = sctrwithin(intrs, nbb[j].ref1, nbb[j].ref2, nbb[j].ref12);
     if(c1 && c2) return(1);
     else
     {
          intrs[0] = -intrs[0];
          intrs[1] = -intrs[1];
          intrs[2] = -intrs[2];
          c1 = sctrwithin(intrs, nba[i].ref1,
                          nba[i].ref2, nba[i].ref12);
          c2 = sctrwithin(intrs, nbb[j].ref1,
                          nbb[j].ref2, nbb[j].ref12);
          if(c1 && c2) return(1);
     }
     return(0);
}

int      sctrwithin(dir, ref1, ref2, ref12)
vector   dir, ref1, ref2, ref12;
{
     vector  c1, c2;
     int     t1, t2;

     cross(c1, dir, ref1);
     if(vecnull(c1, EPS)) return(dot(ref1, dir) > 0.0);
     cross(c2, ref2, dir);
     if(vecnull(c2, EPS)) return(dot(ref2, dir) > 0.0);
     t1 = comp(dot(c1, ref12), 0.0, EPS);
     t2 = comp(dot(c2, ref12), 0.0, EPS);
     return(t1<0 && t2<0);
}
```

**Program 15.9** Sector intersection test.

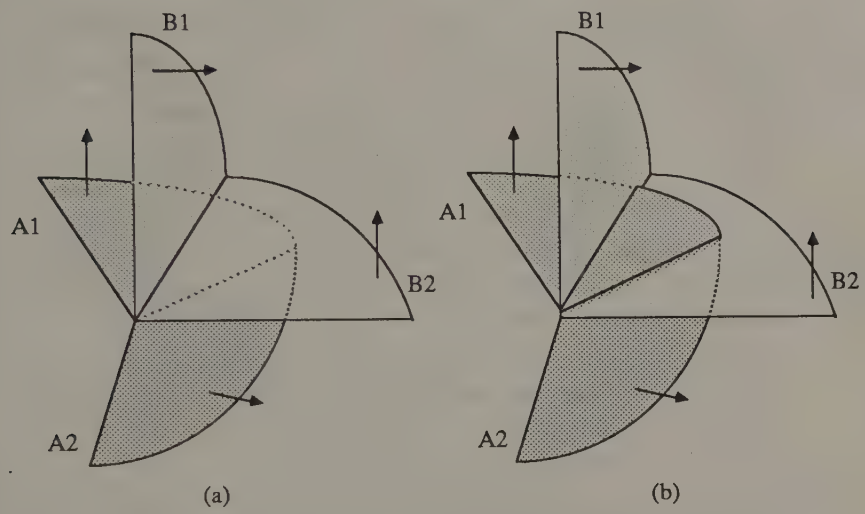



**Figure 15.9** Reclassification on "on"-sectors.

The sector intersection and inclusion tests are implemented in Program 15.9. Most of the code is used to deal with various "special" cases (such as the case that one of the cross products of Equations 15.5 vanishes). The procedure `sectoroverlap` is assumed to check whether two coplanar sectors overlap. This is not elaborated on here.

### Reclassification

The set of potentially intersecting sector pairs found by the sector intersection test will in general include various "special" cases, such as coplanar overlapping sectors and sectors that just touch each other along a bounding edge. Similarly to the classifier for the splitting problem, the next phase of the vertex-vertex classifier will reclassify such "on"-cases as "in" or "out" according to the rules of Equations 15.3.

**Reclassification of "on"-sectors** In strict analogy to the splitting algorithm, the reclassification can proceed by first reclassifying all "on"-sectors and their neighbors.

To come up with suitable reclassification rules, it is useful first to consider an example. Consider the arrangement of four sectors $a_1$, $a_2$ of solid $A$ and $b_1$, $b_2$ of solid $B$ depicted in Figure 15.9(a) that includes two overlapping coplanar sectors $(a_1, b_2)$; arrows indicate the face normals.

After the sector intersection test, we have essentially the following information available in the array *sectors*:

| $A$ | $B$ | Side code for $A$ | Side code for $B$ | intersect bit |
|---|---|---|---|---|
| $a_1$ | $b_1$ | IN-OUT | OUT-ON | 1 |
| $a_1$ | $b_2$ | ON-ON | ON-ON | 1 |
| $a_2$ | $b_2$ | ON-IN | IN-OUT | 1 |

If we intend to calculate the set union, the rules of Equations 15.3 tell us that $a_1$ should be reclassified as lying outside $B$, and $b_2$ as lying inside $A$. That is, the table would read as follows:

| $A$ | $B$ | Side code for $A$ | Side code for $B$ | intersect bit |
|---|---|---|---|---|
| $a_1$ | $b_1$ | IN-OUT | OUT-ON | 1 |
| $a_1$ | $b_2$ | OUT-OUT | IN-IN | 0 |
| $a_2$ | $b_2$ | ON-IN | IN-OUT | 1 |

But this will allow us also to reclassify the information on the neighboring sectors—for instance, as indicated in Figure 15.9, $a_2$ should be treated as if it intersects $b_2$ and its side code should read "OUT-IN". This leads us to the following result:

| $A$ | $B$ | Side code for $A$ | Side code for $B$ | intersect bit |
|---|---|---|---|---|
| $a_1$ | $b_1$ | IN-OUT | OUT-IN | 1 |
| $a_1$ | $b_2$ | OUT-OUT | IN-IN | 0 |
| $a_2$ | $b_2$ | OUT-IN | IN-OUT | 1 |

Now, no "on"-cases remain, and we are done. Observe that as soon as we note that a sector is completely in one side of another, we can reset the intersection indicator.

The sector reclassification rules outlined in the sample case are implemented in Program 15.10. The algorithm simply loops through the array and locates coplanar sector pairs. If one is found, the new classifications for the sector edges are computed according to Equations 15.3. In the second "for"-loop, the neighbor sectors that can be reclassified are located and properly marked. Finally, the coplanar sectors themselves are reclassified.

**Reclassification of "on"-edges** After all coplanar sectors and their neighbors are classified, single "on"-edges may still remain. As indicated in Figure 15.10, there are two cases to be dealt with:

1. The "on"-edge lies within a sector of the other solid (Figure 15.10(a)).
2. The "on"-edge lies on the common edge of two sectors of the other solid (Figure 15.10(b)).

```
void  sreclsectors(op)
int   op;
{
      HalfEdge      *ha, *hb;
      int           i, j, nonopposite, newsa, newsb;
      int           secta, prevsecta, nextsecta;
      int           sectb, prevsectb, nextsectb;

      for(i=0; i<nsectors; i++)
      {
            if(sectors[i].s1a == ON && sectors[i].s2a == ON &&
               sectors[i].s1b == ON && sectors[i].s2b == ON)
            {
                  secta = sectors[i].secta;
                  sectb = sectors[i].sectb;
                  prevsecta = (secta == 0) ? nnba-1 : secta-1;
                  prevsectb = (sectb == 0) ? nnbb-1 : sectb-1;
                  nextsecta = (secta == nnba-1) ? 0 : secta+1;
                  nextsectb = (sectb == nnbb-1) ? 0 : sectb+1;

                  ha = nba[secta].he;
                  hb = nbb[sectb].he;
                  nonopposite = vecequal(ha->wloop->lface->feq,
                                         hb->wloop->lface->feq, EPS);
                  if(nonopposite)
                  {
                        newsa = (op == UNION) ? OUT : IN;
                        newsb = (op == UNION) ? IN : OUT;
                  }
                  else
                  {
                        newsa = (op == UNION) ? IN : OUT;
                        newsb = (op == UNION) ? IN : OUT;
                  }
                  for(j=0; j<nsectors; j++)
                  {
                        if((sectors[j].secta == prevsecta) &&
                           (sectors[j].sectb == sectb))
                              if(sectors[j].s1a != ON)
                                    sectors[j].s2a = newsa;
```

**Program 15.10** Reclassification of "on"-sectors.

```
                if((sectors[j].secta == nextsecta) &&
                   (sectors[j].sectb == sectb))
                       if(sectors[j].s2a != ON)
                             sectors[j].s1a = newsa;
                if((sectors[j].secta == secta) &&
                   (sectors[j].sectb == prevsectb))
                       if(sectors[j].s1b != ON)
                             sectors[j].s2b = newsb;
                if((sectors[j].secta == secta) &&
                   (sectors[j].sectb == nextsectb))
                       if(sectors[j].s2b != ON)
                             sectors[j].s1b = newsb;
                if(sectors[j].s1a == sectors[j].s2a &&
                   (sectors[j].s1a == IN ||
                    sectors[j].s1a == OUT))
                      sectors[j].intersect = 0;
                if(sectors[j].s1b == sectors[j].s2b &&
                   (sectors[j].s1b == IN ||
                    sectors[j].s1b == OUT))
                      sectors[j].intersect = 0;
             }
             sectors[i].s1a = sectors[i].s2a = newsa;
             sectors[i].s1b = sectors[i].s2b = newsb;
             sectors[i].intersect = 0;
          }
       }
}
```

**Program 15.10** Reclassification of "on"-sectors (cont.).

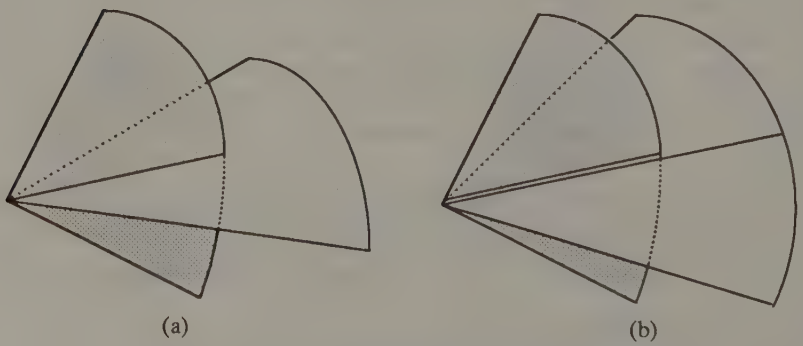



**Figure 15.10** Reclassification of "on"-edges.

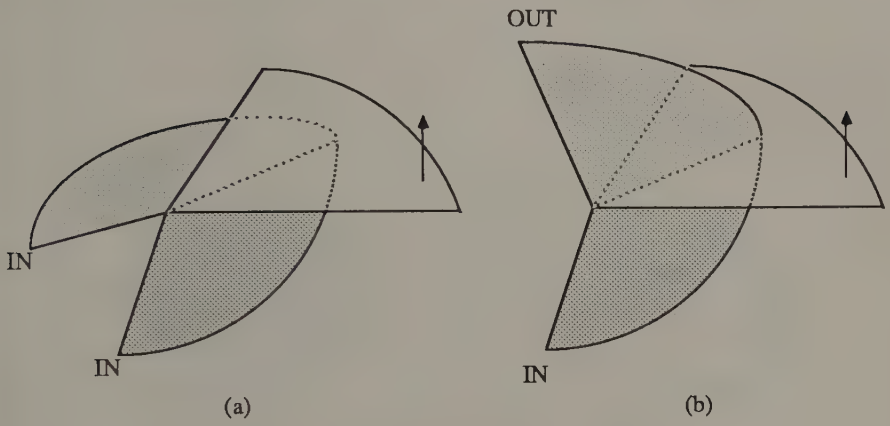



**Figure 15.11** Simple "on"-edges.

The first case can be simply solved by looking at the side codes of the sectors; see Figure 15.11. In the case (a), the sectors are deemed to intersect, and in case (b) no intersection is recorded.

The second case is somewhat more complicated, because we need to look at four sectors. In this case, the existence of an intersection can be determined by sorting the four sectors meeting at the common line angularly around the line; if the sectors appear in a "mixed order," an intersection exists. In Figure 15.12, two arrangements with sectors sharing an edge are depicted on the left, and on the right the same arrangements are shown as viewed along the coincident edge. Clearly, in the first case the sectors intersect, whereas in the second case they do not.

Space does not permit the inclusion of the procedure `srecledges` expected to implement these reclassification rules.

### Insertion of null edges

After reclassification, the final result of vertex-vertex classification is available in the array *sectors* as a sequence of intersecting sector pairs. The final part of the classifier inserts the appropriate null edges into the neighborhood so that the intersection polygons can be generated.

At this point, let us again look at the final result of the sample case of Figure 15.9:

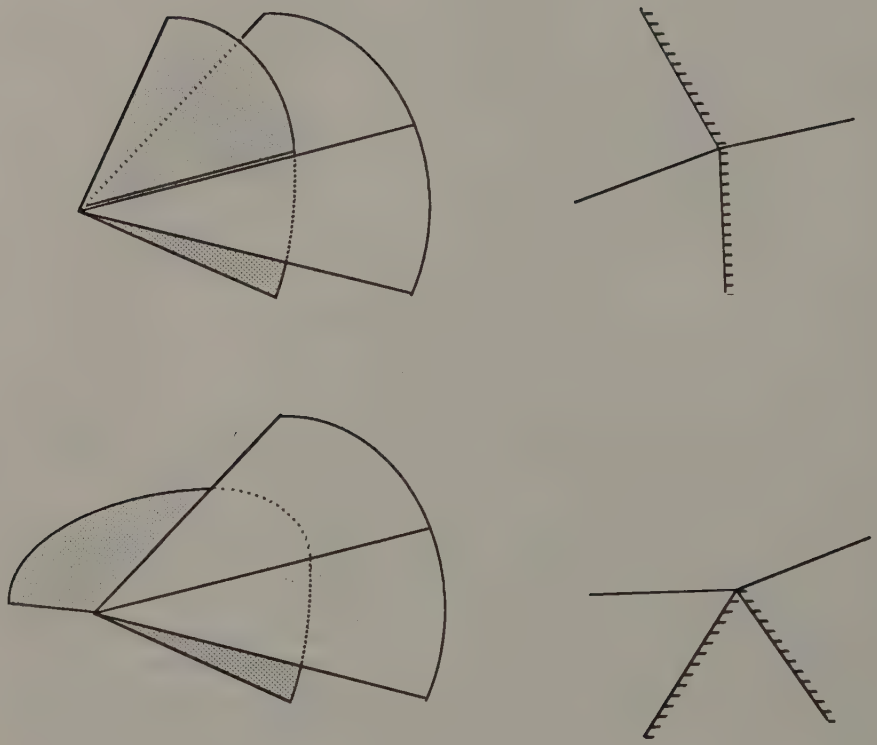


**Figure 15.12** Double "on"-edges.

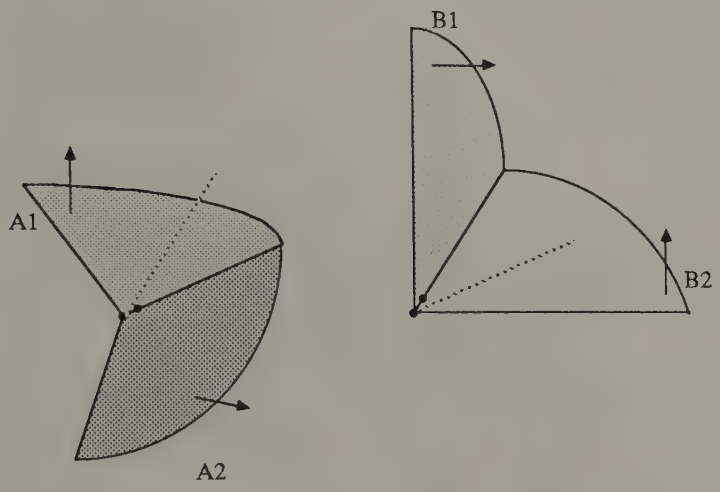



**Figure 15.13** Insertion of null edges.

| $A$ | $B$ | Side code for $A$ | Side code for $B$ | intersect bit |
|---|---|---|---|---|
| $a_1$ | $b_1$ | IN-OUT | OUT-IN | 1 |
| $a_1$ | $b_2$ | OUT-OUT | IN-IN | 0 |
| $a_2$ | $b_2$ | OUT-IN | IN-OUT | 1 |

The middle row can be readily ignored, because its intersection indicator is set to zero. This leaves us with a pair of intersections, and it is easy to determine that the arrangement of Figure 15.13 should result.

The only other case that must be dealt with is the case when a single sector has several intersections. If so, one or the other of the following arrangements will be located:

| $A$ | $B$ | Side code for $A$ | Side code for $B$ | intersect bit |
|---|---|---|---|---|
| $a_1$ | $b_1$ | IN-OUT | OUT-IN | 1 |
| $a_1$ | $b_2$ | OUT-IN | IN-OUT | 1 |
| | | | | |
| $a_1$ | $b_1$ | IN-OUT | OUT-IN | 1 |
| $a_2$ | $b_1$ | OUT-IN | IN-OUT | 1 |

In these cases, the correct action is to insert a "strut" null edge into the sector $a_1$ or $b_1$, respectively.

The insertion of null edges is implemented in Program 15.11. In addition to performing the case analysis discussed, the procedure rearranges

```
void    sinsertnulledges()
{
    HalfEdge    *ha1, *ha2, *hb1, *hb2;
    int         i = 0;

    while(1)
    {
        while(sectors[i].intersect == 0)
            if(++i == nsectors) return;
        if(sectors[i].s1a == OUT)
                ha1 = nba[sectors[i].secta].he;
        else    ha2 = nba[sectors[i].secta].he;
        if(sectors[i].s1b == IN)
                hb1 = nbb[sectors[i++].sectb].he;
        else    hb2 = nbb[sectors[i++].sectb].he;

        while(sectors[i].intersect == 0)
            if(++i == nsectors) return;
        if(sectors[i].s1a == OUT)
                ha1 = nba[sectors[i].secta].he;
        else    ha2 = nba[sectors[i].secta].he;
        if(sectors[i].s1b == IN)
                hb1 = nbb[sectors[i++].sectb].he;
        else    hb2 = nbb[sectors[i++].sectb].he;

        if(ha1 == ha2)
        {
            separ2(ha1, 0);
            separ1(hb1, hb2, 1);
        }
        else if(hb1 == hb2)
        {
            separ2(hb1, 1);
            separ1(ha2, ha1, 0);
        }
        else
        {
            separ1(ha2, ha1, 0);
            separ1(hb1, hb2, 1);
        }
        if(i == nsectors) return;
    }
}
```

**Program 15.11** Insertion of null edges.

```
void            separ1(from, to, type)
HalfEdge        *from, *to;
int             type;
{
        lmev(from, to, ++maxv,
             from->vtx->vcoord[0],
             from->vtx->vcoord[1],
             from->vtx->vcoord[2]);
        if(type == 0) sonea[nedga++] = from->prv->edg;
        else          soneb[nedgb++] = from->prv->edg;
}

void            separ2(he, type)
HalfEdge        *he;
int             type;
{
        HalfEdge        *tmp;

        separ1(he, he, type);
}
```

**Program 15.12** Auxiliaries for the insertion of null edges.

the sectors according to the side codes in order to to give a consistent orientation to the null edges.

The actual insertion of null edges is accomplished by the procedures **separ1** and **separ2** shown in Program 15.12. These procedures also manage the arrays storing the null edges.

## 15.7 JOINING OF NULL EDGES

Let us now consider the procedure **setopconnect** which is expected to combine the null edges generated by the previous steps in order to create the intersection polygons.

This task is obviously similar to the joining problem for the splitting algorithm, with the exception that the intersection polygons are in general not planar figures. Therefore, we cannot expect that the very same algorithm would work for set operations.

Nevertheless, there is a further property of the set operations context that we can exploit. Recall that null edges are always generated in pairs, one into solid $A$ and one into solid $B$. Of course, the intersection polygons

should be constructed in the same fashion in each solid—that is, in the same order through the corresponding null edges.

This property can be used to come up with an algorithm for combining the null edges for set operations. The procedure `scanjoin` of Program 15.13 is a modification of the procedure `canjoin` (Program 14.9) of the splitting algorithm that considers two corresponding null edges at one time, and considers them "joinable" only if both can be joined to a loose end. Current loose ends are maintained in two arrays, one for solid $A$ and one for solid $B$. If the edges cannot be joined, they are added to the set of loose ends; clearly, the updating of the arrays is implemented in a fashion that maintains the correspondence.

Based on `scanjoin`, the joining algorithm for the splitting algorithm can almost mechanically be modified for the set operations. The resulting procedure `setopconnect` is given in Program 15.14. After sorting the two arrays of null edges, the algorithm processes the pairs of corresponding null edges one at a time (by means of procedure `sgetnextnulledge`), and performs the correct actions similarly as the splitting algorithm. The procedures `isloosea`, `islooseb`, `cuta`, and `cutb` are simple modifications of the corresponding procedures `isloose` and `cut` of the spliting algorithm, and we will not elaborate on them.

## 15.8 GENERATION OF THE RESULT

The final phase of the set operations algorithm is expected to actually apply the Equations 15.1 so as to create the final results.

The procedures already developed for the splitting problem are readily useful for the current task as well. The algorithm of Program 15.15 first divides each null face into two intersection faces by means of a `lmfkrh`.

On the basis of the consistent orientation of null edges, the new faces created by `lkfmrh` (`sonfa[nfaca+i]`, `sonfb[nfacb+i]`) are known to belong to the "in"-components (*AinB*, *BinA*) of the boundary classification of $A$ and $B$, while the remaining faces (`sonfa[i]`, `sonfb[i]`) belong to the "out" components.

The algorithm exploits this fact by using the procedure `movefac` of Program 14.12 to move the desired component into the resulting solid *res*. Thereafter, the procedure `cleanup` reconstructs the edge and vertex lists of the result. As the final step, the intersection faces are combined by means of a `lkfmrh` and their edges and vertices merged by means of `loopglue`.

The set difference poses a slight difficulty because the orientation of the "in" component must first be reversed (see Equations 15.1). This is implemented by creating a temporary solid *ainb*, and reversing its orientation

```
static HalfEdge *endsa[30];
static HalfEdge *endsb[30];
static int      nenda;
static int      nendb;

int             scanjoin(hea, heb, reta, retb)
HalfEdge        *hea, *heb, **reta, **retb;
{
        int             i, j;

        for(i = 0; i<nenda; i++)
        {
                if(neighbor(hea, endsa[i]) &&
                   neighbor(heb, endsb[i]))
                {
                        *reta = endsa[i];
                        *retb = endsb[i];
                        for(j=i+1; j<nenda; j++)
                        {
                                endsa[j-1] = endsa[j];
                                endsb[j-1] = endsb[j];
                        }
                        nenda--;
                        nendb--;
                        return(1);
                }
        }
        endsa[nenda++] = hea;
        endsb[nendb++] = heb;
        *reta = NIL;
        *retb = NIL;
        return(0);
}
```

**Program 15.13** Joinability test for set operations.

```
void    setopconnect()
{
        Edge            *nextedgea, *nextedgeb;
        HalfEdge        *h1a, *h2a, *h1b, *h2b;

        nfaca = nfacb = 0;
        ssortnulledges();
        while(sgetnextnulledge(&nextedgea, &nextedgeb))
        {
                if(scanjoin(nextedgea->he1,
                   nextedgeb->he2, &h1a, &h2b))
                {
                        join(h1a, nextedgea->he1);
                        if(!isloosea(mate(h1a)))
                                cuta(h1a);
                        join(h2b, nextedgeb->he2);
                        if(!islooseb(mate(h2b)))
                                cutb(h2b);
                }
                if(scanjoin(nextedgea->he2,
                   nextedgeb->he1, &h2a, &h1b))
                {
                        join(h2a, nextedgea->he2);
                        if(!isloosea(mate(h2a)))
                                cuta(h2a);
                        join(h1b, nextedgeb->he1);
                        if(!islooseb(mate(h1b)))
                                cutb(h1b);
                }
                if(h1a && h1b && h2a && h2b)
                {
                        cuta(nextedgea->he1);
                        cutb(nextedgeb->he1);
                }
        }
}
```

**Program 15.14** Joining algorithm for set operations.

```
void    setopfinish(a, b, res, op)
Solid   *a, *b, **res;
int     op;
{
      int           i, j, inda, indb;

      inda = (op == INTERSECT) ? nfaca : 0;
      indb = (op == UNION)     ? 0 : nfacb;
      *res = (Solid *) new(SOLID, NIL);
      for(i=0; i<nfaca; i++)
      {
            sonfa[nfaca+i] = lmfkrh(sonfa[i]->floops->nextl,
                  ++maxf);
            sonfb[nfacb+i] = lmfkrh(sonfb[i]->floops->nextl,
                  ++maxf);
      }
      if(op == DIFFERENCE)
            revert(b);
      for(i=0; i<nfaca; i++)
      {
            movefac(sonfa[i+inda], *res);
            movefac(sonfb[i+indb], *res);
      }
      cleanup(*res);
      for(i=0; i<nfaca; i++)
      {
            lkfmrh(sonfa[i+inda], sonfb[i+indb]);
            loopglue(sonfa[i+inda]);
      }
}
```

**Program 15.15** Generation of the result.

by means of the procedure revert left as an exercise to the reader. After this, merge will do the right thing.

Observe that the parts of $A$ and $B$ not required in the final result will remain in the original solids. This means that the algorithm could be extended to calculate both $A \cup B$ and $A \cap B$, or $A \setminus B$ and $B \setminus A$, simultaneously.

## 15.9 FINAL REMARKS

As suggested by Requicha and Voelcker [96], the techniques for classification and neighborhood analysis developed for boundary evaluation of CSG models are applicable to evaluation Boolean operations of BR models as well. Indeed, the algorithm described can be regarded as a particular way to implement the theoretical procedures Requicha and Voelcker outline in their article. However, while they concentrate on explicitly represented face and edge neighborhoods, our algorithm is based on implicitly represented vertex neighborhoods. Moreover, our algorithm is much more economical by exploiting the adjacency information usually available in "data-rich" boundary representations such as the winged-edge representation, and by avoiding point-in-polyhedron tests completely (except for the case that the surfaces of the solids no not intersect at all).

The algorithm does not extend directly to objects with curved faces or to nonmanifold objects. Curved faces may intersect each other in a way that cannot be reduced to vertex-vertex coincidences (consider spheres, for instance). Set operations of nonmanifolds would require the classification of several vertex neighborhoods from one object against several neighborhoods from the other, instead of the one-against-one case handled by this algorithm.

## BIBLIOGRAPHIC NOTES

As noted in the introduction of this chapter, the literature of algorithms for Boolean set operations is meager and often quite inaccurate. Nevertheless, some traces can be given to the interested reader.

The algorithm described can be viewed as an implementation of the methods described in [73]; it is an offspring of earlier algorithms described in [117,79]. The idea of reclassification of "on"-entities appears in various levels of explicitness in several publications, such as [27,135].

As presented, the algorithm is slow. A practical Boolean set operations algorithm should definitely employ various methods for speeding up the computation by means of enclosing boxes and similar aids. A careful implementation of the

edge-face comparison of Section 15.4 can speed the whole algorithm up by a factor of 10 or so, but the quadratic basic behavior still remains.

However, as noted by Requicha and Voelcker, the worst-case analysis of the computational efficiency for set operations is not very informative as for the practical speed of the algorithm, and advanced geometric search techniques can be applied to it so as to exploit the typical geometric locality of the computation. An example of this is described in [79].

Figures 15.2 through 15.12 have been previously published in the article "Boolean operations of 2-manifolds through vertex neighborhood classification" by M. Mäntylä in *Transactions on Graphics*, Volume 5, Number 1. ©1986, Association for Computing Machinery, Inc. Reprinted by permission.

## PROBLEMS

15.1. Program 15.1 does not deal properly with the case that one solid is completely within the other. Generalize the algorithm so that it will produce the correct results even in these cases.

*Hint:* You will need a procedure for testing the inclusion of a vertex within a solid.

15.2. Implement the procedure

```
void    maximize_faces(s)
Solid   *s;
```

that removes all inessential edges (i.e., edges that separate two coplanar faces, or occur within just a single face) of solid *s*.

*Hint:* Observe that you have to deal with special cases such as a face lying completely within another.

15.3. Reimplement the algorithm for edge-face comparison so that the recursion is eliminated.

*Hint:* Gather all intersections of the query edge into an auxiliary data structure. After all comparisons, sort the intersections along the edge, and process them in one sweep.

15.4. Implement the vertex-face classifier for set operations by modifying Programs 14.3–7 as suggested in Section 15.6.1. (With some modifications to the splitting algorithm, the *same* code could actually be used.)

15.5. Implement the procedure `sectoroverlap((h1, h2))` that tests whether two coplanar sectors `h1` and `h2` overlap.

15.6. Implement the procedure `revert(s)` that reverses the orientation of its argument solid *s*.

*Hint:* This task is best implemented through direct manipulation of the data structure. You will have to reverse the orientation of all loops and face equations.

15.7. Extend the algorithm to calculate both $A \cup B$ and $A \cap B$, or $A \setminus B$ and $B \setminus A$, simultaneously.

15.8. On the basis of the three-dimensional set operations algorithm presented, write a two-dimensional set operations algorithm that works on coplanar laminae.

*Hint:* Your algorithm can start by sweeping the laminae so as to create three-dimensional solids, and computing their Boolean combination. After that, the procedure `maximize_faces` of Problem 15.2 and the procedure `unsweep` of Problem 12.4 on page 216 can be used to generate the desired result.

15.9. Test your understanding: Would the set operations algorithm correctly compute $A \setminus B$ as the set intersection of $A$ and reversed $B$?

# Chapter 16

# UNDOING

## 16.1 MOTIVATION

To err is human. A practical software system must take into account the fact that its users will make various sorts of mistakes during the use of the system. If no protection against errors is provided, the system is very likely to be abandoned.

To a certain extent, it is possible to avoid errors by introducing shielding mechanisms that protect potentially "dangerous" operations. Unfortunately, shields irritate experienced users while not necessarily helping the naive users.

A better solution is to augment error avoidance techniques with proper error recovery techniques that allow the user to nullify the effects of an erroneous command. The most general form of error recovery is the *undo command* that can remove the effect of any previous operation (except for itself).

Undoing comes in many flavors. Sometimes only limited undoing that can recall some fixed number of previous states is provided. While users will not very often undo more than one or two commands, the full power of undoing only becomes available if an unlimited undo operation that can remove the effect of any number of previous operations is possible. It gives the user the freedom of experimenting with alternatives with the knowledge that a wrong path can always be traced back.

A generalization of the undo command is the *interrupt mechanism* that allows the undoing of an partially executed operation. Suppose the user has started a time-consuming operation, only to notice that it is somehow erroneous. The fact that the operation can be undone does not relieve the user from the pain of first having to wait for the end of the erroneous

operation. In such cases, some sort of an interrupt mechanism that allows the user to interrupt the operation and resume the previous state should be supported.

A properly implemented interrupt mechanism can be used not only by the user, but also by the algorithms of the system itself. Sometimes it is not practical to check for the correctness of the input data to an algorithm until some real computation that alters the state of the system has been done. If the algorithm is unable to continue, it should roll back all changes it has caused. Clearly, an interrupt operation will do the right thing. Going further, an interrupt operation can even be used to protect the system against unexpected internal error conditions, provided that underlying operating system provides support for something equivalent to the signal handlers of the UNIX operating system.

This chapter describes the design and implementation of an unlimited undoing and interrupting mechanism for GWB. In addition to its practical value, the operation will also highlight the real power of the strictly layered architecture of GWB and of the Euler operators. We shall also show how the undo operation can be utilized beyond the mere error recovery.

## 16.2 OVERVIEW

A prime example of the necessity of undoing is given by real time data base manipulation applications that must be designed so as to preserve the *transaction integrity*. For instance, the sequence of operations required for transfering $100.00 from the bank account of my publisher to my personal account must either be processed completely, or not at all.

The typical solution is based on an *undo log* that records all individual changes to the data base. If the operation must be canceled for some reason, the log can be examined so as to nullify all changes. (For maximal security, the data base is usually modified only after all changes have been written into the log, and the application has issued a special `commit` command.)

Observe that the undo log approach is applicable in this case because the effect of the banking transaction can be adequately represented as a relatively short sequence of simple assignment operations, where some old piece of data is replaced by a new one. Then a sequence of log items of the form

```
<transaction-id, old value, new value>
```

will be sufficient.

For more general computations, the assignment-based undo log approach is usually not applicable because of the prohibitively large bulk

of assignments involved. Clearly, this applies to solid modeling as well: solid modeling algorithms frequently modify quite large amounts of data during a single user-level operation.

To make the undo log approach feasible for applications such as solid modeling, we should replace the simple assignments by some much more powerful basic operations. The operations should be powerful enough to represent the effect of the computation, yet simple enough to have a concise representation in the undo log.

Fortunately for us, Euler operators fill this specification almost exactly. By the theoretical results of Chapter 10, we know that Euler operators are powerful enough to describe all meaningful manipulations of GWB data structures. Moreover, the algorithms of Part Two were implemented (almost) solely by means of the Euler operators. Finally, it is possible to use vertex, face, and solid identifiers to represent Euler operators efficiently in an undo log.

This leads us to the following design for an undo facility for GWB:

1. Design an undo log that can store arbitrary sequences of Euler operators and other basic operations of GWB.

2. Provide all Euler operators with the capability of maintaining the undo log. That is, in addition to manupulating the GWB data structures as usual, each operator is responsible for storing such Euler operators that can undo its effect into the undo log.

3. Of course, most interesting (user level) operations cause more than one Euler operator to be executed. Therefore, we need appropriate facilities for managing modeling transactions consisting of a (potentially long) sequence of basic operations.

The following sections will elaborate on these items.

## 16.3 AN UNDO LOG DATA STRUCTURE

Most algorithms of GWB of Part Two use the low-level Euler operators which are parameterized in terms of pointers to the GWB data structure. For the undo log, we need an equally powerful set of operators parameterized by means of solid, face, and vertex identifiers. Fortunately, the set of high-level Euler operators as defined in Section 11.5.2 can be used. For reference, the high-level operators and their parameterizations are given in Figure 16.1.

With this parameterization, the undo log data structure becomes very straightforward. As shown in Program 16.1, we can simply use a linked list

```
Id      s, f, f1, f2, v, v1, v2, v3, v4;
float   x, y, z;

Solid   mvsf(s, f, v, x, y, z);
int     kvsf(s);

int     mev(s, f1, f2, v1, v2, v3, v4, x, y, z);
int     kev(s, f, v1, v2);

int     mekr(s, f, v1, v2, v3, v4);
int     kemr(s, f, v1, v2);

int     mef(s, f1, v1, v2, v3, v4, f2);
int     kef(s, f1, v1, v2);

int     kfmrh(s, f1, f2);
int     mfkrh(s, f1, v1, v2, f2);
```

**Figure 16.1** The parameterization of high-level Euler operators.

of records consisting of an operation code that identifies the Euler operator, a field for the solid identifier, six other integer fields that are used to store the integer parameters of Euler operators, and four floating-point fields that are used to store the coordinate information. (Of course, a variable-length record could also be used.) The procedure **addeulerop** is intended for adding one Euler operator to the log; its implementation is left to the reader.

Program 16.1 also includes a few global variables which will be used later in this chapter. The variable *OpHead* will point at the head of the undo log (that is, the most recent log record). The variable *OpCount* will store the current number of log records. The global variable *Generatelog* will be used to turn undo logging on and off; if *Generatelog* is zero, no log records will be added to the log.

## 16.4 UNDOABLE EULER OPERATORS

Let us now look into the task of incorporating the management of the undo log into the low-level Euler operators. Fortunately, in most cases the changes required in the program codes are almost trivial.

As a simple example, the modified code of the `lmev` operator is shown in Program 16.2. The only change from the original version of Program 11.6

```
# define        MVSF    0
# define        KVSF    1
# define        MEV     2
# define        KEV     3
# define        MEF     4
# define        KEF     5
# define        KEMR    6
# define        MEKR    7
# define        KFMRH   8
# define        MFKRH   9

typedef struct eulerop  EulerOp;
struct eulerop
{
        short   opcode;
        Id      solidno;
        Id      ip1, ip2, ip3, ip4, ip5, ip6;
        double  fp1, fp2, fp3, fp4;
        EulerOp *opnext;
};
EulerOp *OpHead;
int     OpCount;
int     Generatelog;

void    addeulerop(opcode, solidno, ip1, ip2, ip3,
                   ip4, ip5, ip6, fp1, fp2, fp3, fp4)
short   opcode;
Id      solidno;
Id      ip1, ip2, ip3, ip4, ip5, ip6;
double  fp1, fp2, fp3, fp4;
{
        ...
}
```

**Program 16.1** Undo log data structures and operations.

```
void            lmev(he1, he2, vn, x, y, z)
HalfEdge        *he1, *he2;
Id              vn;
double          x, y, z;
{
        HalfEdge        *he;
        Vertex          *newvertex;
        Edge            *newedge;

        if(Generatelog)
                addeulerop(KEV,
                        he1->wloop->lface->fsolid->solidno,
                        he1->wloop->lface->faceno,
                        he1->vtx->vertexno,
                        vn,
                        0, 0, 0, 0.0, 0.0, 0.0, 0.0);

        newedge = (Edge *) new(EDGE, he1->wloop->lface->fsolid);
        newvertex = (Vertex *) new(VERTEX,
                he1->wloop->lface->fsolid);
        newvertex->vcoord[0] = x;
        newvertex->vcoord[1] = y;
        newvertex->vcoord[2] = z;
        newvertex->vcoord[3] = 1.0;
        newvertex->vertexno = vn;

        he = he1;
        while(he != he2)
        {
                he->vtx = newvertex;
                he = mate(he)->nxt;
        }

        addhe(newedge, he2->vtx, he1, MINUS);
        addhe(newedge, newvertex, he2, PLUS);

        newvertex->vedge = he2->prv;
        he2->vtx->vedge = he2;
}
```

**Program 16.2** The undoable lmev operator.

is the guarded call of the procedure addeulerop. In this case, the operator kev is stored according to the parameterization of Figure 16.1. (The reader should verify that this operator can indeed undo the effect of lmev.)

Unfortunately, not all cases are as simple. It turns out that making the operator lkef undoable poses the most difficult challenge. This is so because the faces joined by the lkef operator may have several loops, which will all become attached into a single face after the operation. Therefore, the undo operation must be capable of rearranging the loops back into their original faces.

An undoable implementation of the lkef operator is shown as Program 16.3. The solution is to include the "ring move" operator ringmv into the scope of undoing, and insert the appropriate operations into the log before inserting the undo record for the lkef itself. Now, if the lkef operation is undone, the ringmv operations will be executed after the lmef operation. They will move the loops that used to belong to the face *f2* recreated back to their proper places.

## 16.5 OTHER UNDOABLE OPERATIONS

Most operations of GWB are carried out by means of Euler operators. Hence, they can be undone by making the Euler operators undoable. However, some algorithms of GWB also manipulate solids directly, without the Euler operators. These include major parts of GWB such as the splitting and the Boolean set operations algorithms. To make them undoable, we must extend the undo facility to cover these manipulations also.

The need for manipulating solids by operations other than Euler operators stems from the fact that splitting and set operations must be capable of dividing a solid into two solids. This task is beyond the scope of the Euler operators used in this text.

Looking back at the final steps of the set operations algorithm (Program 15.15), we can see that the division is carried out by means of the procedure `movefac(f, s)` that moves the face $f$ and all faces recursively connected with it to solid $s$. For reference, the relevant part of Program 15.15 is shown as Program 16.4.

Fortunately, to make movefac undoable is straightforward. We simply have to store an undo record that requests the connected set of faces of $f$ to be moved back to their original solid, which is available as `f->fsolid`. The revised version of movefac that implements this is shown in Program 16.5.

Program 16.4 also includes the procedure revert that complements its argument solid. Because it is its own inverse, its undoing is trivial and left to the reader. In addition, the set operations algorithm uses the procedure

```
void        lkef(h1, h2)
HalfEdge    *h1, *h2;
{
    Face        *f1, *f2;
    Loop        *l, *l1, *l2;
    HalfEdge    *he;
    Vertex      *v3;

    l1 = h1->wloop;
    f1 = l1->lface;
    l2 = h2->wloop;
    f1 = l2->lface;
    if(Generatelog)
    {
        for(l = f1->floops; l = l->nextl; l != NIL)
        {
            if(l != l2)
                addeulerop(RINGMV, f1->fsolid->solidno,
                    l->ledg->vtx->vertexno,
                    l->ledg->nxt->vtx->vertexno,
                    f1->faceno, f2->faceno,
                    (l == f1->flout ? 1 : 0),
                    0, 0.0, 0.0, 0.0, 0.0);
        }
        addeulerop(MEF, f1->fsolid->solidno, f1->faceno,
            h2->vtx->vertexno, h2->nxt->vtx->vertexno,
            h1->vtx->vertexno, h1->nxt->vtx->vertexno,
            f2->faceno, 0.0, 0.0, 0.0, 0.0);
    }

    /* the rest is the ordinary lkef code */
    ...
}
```

**Program 16.3** The undoable `lkef` operator.

```
if(op == DIFFERENCE)
      revert(b);
for(i=0; i<nfaca; i++)
{
      movefac(sonfa[i+inda], *res);
      movefac(sonfb[i+indb], *res);
}
cleanup(*res);
for(i=0; i<nfaca; i++)
{
      lkfmrh(sonfa[i+inda], sonfb[i+indb]);
      loopglue(sonfa[i+inda]);
}
```

**Program 16.4** Division of a solid.

updmaxnames that rearranges the face and vertex identifiers of its argument solid; it must also be made undoable.

## 16.6 TRANSACTION MANAGEMENT

The undo facilities developed so far form only the lowest level of the undo functionality. They must be appended with user-level operations to make undoing available to all GWB algorithms.

First, we need a facility for marking the boundaries of a single modeling transaction that must be treated as a unit of undoing. Typically, the processing caused by one interactive user command would form a single transaction.

The design of Program 16.6 uses a simple cyclically organized array *TransactionLog* for marking transaction boundaries. When a transaction is started (procedure `BeginTransaction`), the current number of Euler operators stored in the undo log is saved into the array, and the indices *TransactionTop*, *TransactionBottom* are updated appropriately. The static variable *TransactionDepth* is used to keep track of nested calls to *BeginTransaction* and *EndTransaction*. In this implementation, no log records are made for the inner transactions.

A complete transaction can be undone in its entirety by means of the procedure `UnDoTransaction`. To actually undo a single Euler operator (or any other undoable basic operation), the procedure calls the procedure undoop given in Program 16.7. Of course, the procedure must switch undo logging off.

```
void    movefac(f, s)
Face    *f;
Solid   *s;
{
        if(Generatelog)
                addeulerop(MOVEFAC, s->solidno,
                        f->faceno, f->fsolid->solidno,
                        0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0);
        domovefac(f, s);
}

void    domovefac(f, s)
Face    *f;
Solid   *s;
{
        Loop            *l;
        HalfEdge        *he;
        Face            *f2;

        dellist(FACE, f, f->fsolid);
        addlist(FACE, f, s);
        l = f->floops;
        while(l)
        {
                he = l->ledg;
                do
                {
                        f2 = mate(he)->wloop->lface;
                        if(f2->fsolid != s)
                                movefac(f2, s);
                }
                while((he = he->nxt) != l->ledg);
                l = l->nextl;
        }
}
```

**Program 16.5** Undoable division of a solid.

```
int     TransactionLog[MAXUNDO];
int     TransactionTop = 0;
int     TransactionBottom = 0;
int     TransactionDepth = 0;

void    BeginTransaction()
{
        int     i;

        if(TransactionDepth++ > 0) return;
        TransactionLog[TransactionTop] = OpCount;
        TransactionTop++;
        if(TransactionTop == MAXUNDO)
                TransactionTop = 0;
        if(TransactionTop == TransactionBottom)
        {
                TransactionBottom++;
                if(TransactionBottom == MAXUNDO)
                        TransactionBottom = 0;
        }
        Generatelog = 1;
}

void    EndTransaction()
{
        if(TransactionDepth > 0)
                TransactionDepth--;
}

void    UnDoTransaction()
{
        int     nops;

        if(UnDoTop == UnDoBottom) return;
        UnDoTop--;
        if(UnDoTop < 0)
                UnDoTop = MAXUNDO;
        nops = UnDoRecs[UnDoTop].modelops;
        while(OpCount > nops)
                undoop();
        TransactionDepth = 0;
}
```

**Program 16.6** Transaction log management.

```
Solid   *undoop()
{
        Solid    *s;
        EulerOp  *op;
        int      Generatelog_save;

        Generatelog_save = Generatelog;
        Generatelog = 0;
        op = OpHead;
        if(op)
        {
                OpHead = op->opnext;
                s = applyop(op);
                free(op);
                OpCount--;
        }
        else    s = NIL;
        Generatelog = Generatelog_save;
        return(s);
}
```

**Program 16.7** Undoing single operations.

Finally, the procedure applyop of Program 16.8 interprets a single undo log record and executes the appropriate operations. For completeness, we give the whole procedure, although we have not discussed all operations covered by the procedure.

## 16.7 APPLICATIONS OF UNDOING

The undoing facility developed in the preceding sections is useful for supporting an interactive undo command invoked by a user. Besides this, there are many other ways of using the undoing machinery.

### 16.7.1 Error Rollback

A straightforward application of the undoing functionality is the rollback of modifications after a runtime error has been encountered within an algorithm.

Suppose, for instance, that the set operations algorithm notes an irrevocable error after considerable changes to the objects have already been made. In this case, the algorithm can easily use the undoing for canceling all modifications it has made to the argument solids up to the point of the error. To avoid interference with the user-level transaction management, the algorithm can itself mark the initial state by including the following C statements before any other processing:

```
int Generatelog_save, OpCount_save;

Generatelog_save = Generatelog;
Generatelog = 1;
OpCount_save = OpCount;
```

By this, the algorithm can use undo facilities whether undoing is enabled or not.

If an error is encountered, changes performed so far can be rolled back as follows:

```
while(OpCount > Opcount_save)
      undoop();
Generatelog = Generatelog_save;
```

Of course, this technique is not restricted to error recovery, but may also be used for more general purposes. The next section will give an extended example of this.

```
Solid        *applyop(op)
EulerOp      *op;
{
    Solid    *s, *sh;

    if(op->solidno > -1 && op != MVFS)
        s = ssolid(op->solidno);
    s2 = s;
    switch(op->opcode)
    {
        case RINGMV:
            ringmv(s, op->ip1, op->ip2, op->ip3, op->ip4, op->ip5);
            break;
        case REVERT:
            revert(s);
            break;
        case MODIFYNAMES:
            modifynames(s, op->ip1, op->ip2);
            break;
        case MOVEFAC:
            movefac(fface(s, op->ip1), ssolid(op->ip2));
            redo(s);
            redo(ssolid(op->ip2));
            break;
        case MVFS:
            s2 = mvfs(op->solidno, op->ip1, op->ip2,
                      op->fp1, op->fp2, op->fp3);
            break;
        case KVSF:
            kvfs(op->solidno);
            s2 = NIL;
            break;

        /* other operators */
        ...
    }
    return(s2);
}
```

**Program 16.8** Applying undo log records.

## 16.7.2 File Operations of Solids

As described so far, GWB does not include procedures for storing an instance of the half-edge data structure onto secondary storage (disk), or for reading a previously stored data structure back. In this section we shall show how the undo machinery can be used to fill this gap.

### Approaches to File Representations of Solids

In principle, the designer of filing operations for a boundary modeler such as GWB has several alternative strategies to choose from. First, it would be possible to store the construction sequence used to generate the solid in the first place. While this approach usually leads to rather concise data base formats, the computational cost of re-evaluating the model can be large if procedures such as the Boolean set operations are used in the construction. Moreover, in order to make data transfer between several programs possible, all programs would have to be capable of evaluating all possible construction operations. Still worse, as the operations stored are peculiar to the particular modeling system, the representation cannot support transfer of information to another modeler.

Clearly, a file format which is independent of the construction history of the solid avoids these problems. One alternative is to directly convert the run-time data structure of the solid into a suitable "pointer-free" external format. To regenerate the object, another conversion back to central storage format is needed. Alternatively, relocatable data structures can be used to represent the solid model in the identical fashion both in central storage and in secondary storage. In this case, transfer of data between central and secondary storage could be handled by a storage management utility. The drawback of a direct conversion is the inherent complexity of modifying a "pointer-rich" data structure such as the half-edge data structure into a file format. Moreover, the external data structure easily becomes voluminous.

The third alternative is the indirect conversion that tries to avoid the problems of direct conversion by mapping the internal data structures to a format particularly useful for file storage. The problem that remains in this approach is the selection of a format that exhibits a good balance between the complexity of generation and retrieval and the size of the external representation.

### Solid Inversion

The approach to be followed here can be viewed as a combination of the construction history approach and the indirect conversion approach.

To grasp the basic idea, it is useful to recall the analogy of Chapter 8 between computer languages and Euler operators. This analogy considers the Euler operators an "assembler" on top of which all higher-level operations "execute." In this view, all models of GWB are the result of executing a sequence of Euler operators.

According to the construction history approach, the model could be stored by storing the sequence of operators used to generate the model. The problem that remains is that the sequence may be much longer than necessary. If, say, one of the high-level machines that execute on top of Euler operators is the set operations algorithm, large numbers of faces, edges, and vertices are created and thrown away.

Ideally, we need a "disassembler" that can examine an instance of the half-edge data structure and create a "short" sequence of Euler operators capable of creating it. Then, we could use the "short" as a representation of the object in secondary storage.

At this point, the undo mechanism comes into play. Imagine that we have a procedure capable of erasing a solid by means of the Euler operators. Because the procedure uses Euler operators only, it is undoable—that is, after the procedure has finished, the undo log will contain a sequence of operators that will recreate the solid if executed. Hence, we have indeed found a "disassembly" of the original solid!

We can now immediately give an outline of the file generation algorithm in Program 16.9. The two procedures `remedg` and `remfac` constitute the solid removal algorithm; they are described in the next section. After that, the usual undo loop is applied, except that all operators are first written into a file by means of the procedure `writeop`. This procedure is not elaborated on here.

### Solid Removal Algorithm

The problem of removing an instance of the half-edge data structure by means of Euler operators was already encountered in Chapter 9 in the context of proving the completeness of plane model manipulation operations. In fact, the algorithm to be presented can be viewed as a mechanization of the proof of Theorem 9.2, page 141.

The removal is best implemented in two phases. First, all edges of the solid are removed (procedure `remedg` in Program 16.9). After that, the solid will have been reduced to a rather peculiar skeletal object with one lone vertex for each connected component of edges in the original solid. This object is removed in the second phase (procedure `remfac`).

The task of the procedure `remedg` is well represented in Figure 9.2, page 142. Analogously to the proof of Theorem 9.2, the procedure can

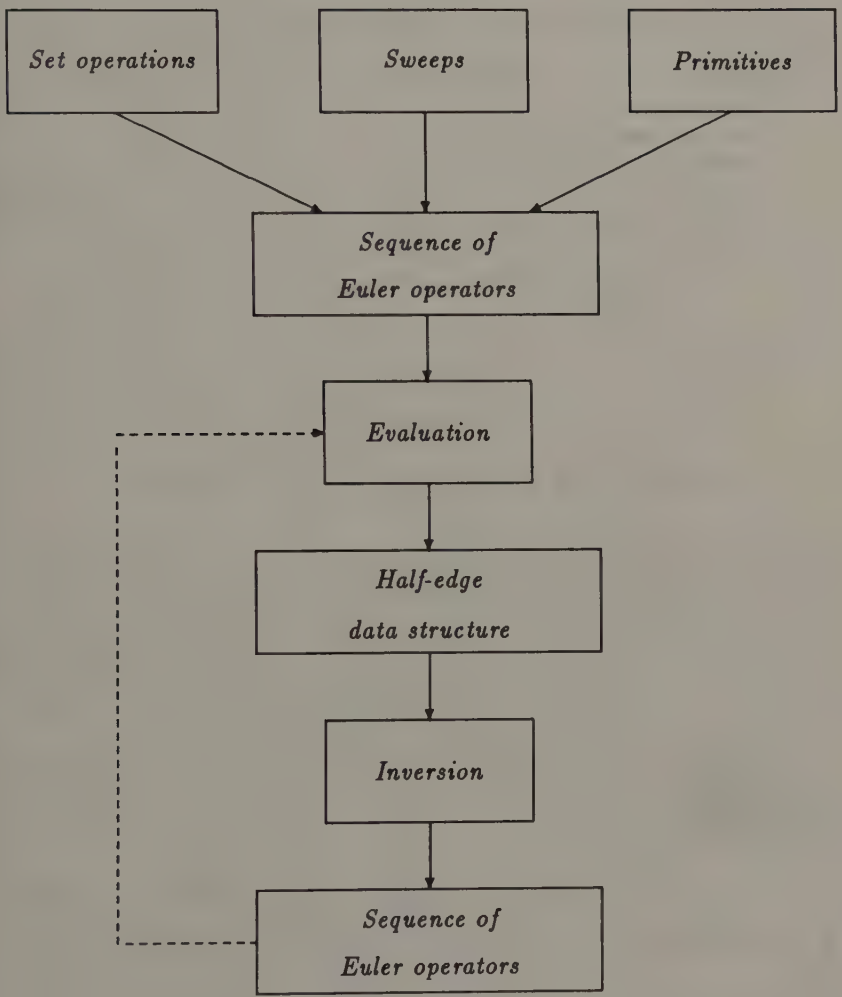



**Figure 16.2** The inversion approach.

```
void    invert(s, file)
Solid   *s;
FILE    *file;
{
        int     Generatelog_save, OpCount_save;

        Generatelog_save = Generatelog;
        Generatelog = 1;
        OpCount_save = OpCount;
        remedg(s);
        remfac(s);
        while(OpCount > Opcount_save)
        {
                writeop(OpHead, file);
                undoop();
        }
        Generatelog = Generatelog_save;
}
```

**Program 16.9** Outline of the inversion algorithm.

work by considering all edges of the solid in turn, and deciding with a simple case analysis which Euler operator to use for removing the edge.

An implementation of the case analysis is outlined in Program 16.10. The procedure `nextedge` is assumed to supply a pointer to the next edge $e$ of the solid $s$ to be removed. The algorithm examines the loops and faces of $e$, and applies the appropriate Euler operator.

The order in which the edges are removed has significance to the length of the resulting operator sequence. Fortunately, the rules for achieving the shortest sequence are simple:

1. If these is an edge such that it can be removed with `lkev`, remove it.

2. Otherwise, if there is an edge that can be removed with `lkef`, remove it.

3. Otherwise, remove any edge.

These rules could be implemented in the procedure `nextedge`.

After the edges have been removed, each shell (connected set of faces) of the original solid has shrunk to a single face, consisting of a one-vertex loop for each original connected set of edges of that shell. The face removal phase

```
void    remedg(s)
Solid   *s;
{
        Edge            *e;
        Loop            *l1, *l2;

        while(e = nextedge(s))
        {
                l1 = e->he1->hloop;
                l2 = e->he2->hloop;
                if(l1 == l2)
                {
                        if(e->he1->nxt == e->he2)
                                lkev(e->he2, e->he1);
                        else if(e->he2->nxt == e->he1)
                                lkev(e->he1, e->he2);
                        else    lkemr(e->he1, e->he2);
                }
                else if(l1->lface != l2->lface)
                        lkef(e->he1, e->he2);
                else    lmfkrh(l1->lface,
                            (l1 == l1->lface->flout) ? l1 : l2);
        }
}
```

**Program 16.10** Removal of edges.

will reduce each shell to a one-vertex "skeletal" shell, and will combine the shells with each other until a single one-vertex shell remains.

The procedure `remfac` that implements this manipulation is shown as Program 16.10. The algorithm first joins the "inner" loops of all remaining faces with their corresponding "outer" loops by means of `lmekr`'s, and deletes the ring vertex with `kev`'s. In the last `while` loop, all faces except one are removed by first reducing them to rings with `kfmrh`'s, and applying the same operators as above.

### Reading an Inversion

The inversion algorithm creates a file consisting of a sequence of Euler operators as written by the procedure `writeop`. In practice, `writeop` may well use variable length records to represent Euler operators in secondary storage—say, coordinate data is written only if *opcode* = MEV or MVFS.

Whatever encoding tricks are used in the file, a procedure `readop` must exist such that it is capable of reconstructing Euler operators into an undo log record. Based on `readop`, it is easy to write a procedure that reads an inversion file and returns a pointer to the solid read. It is outlined in Program 16.11.

Ordinarily inversions would be written and read in binary format for greater space efficiency. However, if inversions must be transmitted from one computer to another, binary files can be replaced by ASCII files just by modifying the procedures `writeop` and `readop`.

### Notes

The inversion algorithm can be viewed as a constructive proof of Theorem 9.2: every correct instance of the half-edge data structure can be created with a sequence of Euler operators. In fact, because the algorithm works without the operators KVFS, KEV, or KEF, it shows the slightly stronger result that none of these "negative" operators are needed in such a sequence.

Due to this fact, the performance of the algorithm can easily be analyzed. Because no "negative" operators appear, the operator counts of Equations 9.5 will hold for all but KEMR and MEKR. That is, in the absence of rings and holes, a solid whose with $v$ vertices, $e$ edges, and $f$ faces that form a connected surface is represented with exactly with the following numbers of operators:

| | |
|---|---|
| $v-1$ | MEV's |
| $f-1$ | MEF's |
| 1 | MVFS |

```
void    remfac(s)
Solid   *s;
{
        Face            *f1, *f2;
        Loop            *out, *ring;

        f1 = s->sfaces;
        while(f1)
        {
                out = f1->flout;
                ring = f1->floops;
                while(f1->floops->nextl)
                {
                        if(ring != out)
                        {
                                lmekr(out->lhedg, ring->lhedg);
                                lkev(out->lhedg, ring->lhedg);
                                ring = f1->floops;
                        }
                        else    ring = ring->nextl;
                }
                f1 = f1->nextf;
        }
        f1 = s->sfaces;
        while(f2 = f1->nextf)
        {
                lkfmrh(f1, f2);
                lmekr(f1->floops->lhedg, f1->floops->nextl->lhedg);
                lkev(f1->floops->lhedg, f1->floops->nextl->lhedg);
        }
}
```

**Program 16.11** Removal of faces.

```
Solid   *iread(filename)
char    *filename;
{
        FILE            *file;
        Solid           *s;
        EulerOp         op;

        file = fopen(filename, "r");
        while(readop(file, &op))
                s = applyop(s, op);
        fclose(file);
        return(s);
}
```

**Program 16.12** Reading an inversion.

Hence, a total of

$$v + f - 1$$

operators would be required in this simple case.

In the general case, rings and holes complicate the counting, because both KEMR, KFMRH and their inverses MEKR, MFKRH are needed in the general case. Observe, however, that MEKR's will only needed in objects that have holes, and that at most one MEKR for each hole is needed. Hence, the number $m$ of MEKR's required in an inversion must be less than $h$, the number of holes.

Let $v$, $e$, $f$, $s$, $h$, and $r$ denote the number of vertices, edges, faces, shells, holes, and rings in the solid, and let $m$ be the number of MEKR'S required, $0 \leq m \leq h$. Then the number of operators required in an inversion can be counted follows:

1. By examination of the algorithm, it is clear that 1 MVFS, $s - 1$ MFKRH's, and $h$ KFMRH's are always generated. These operators generate $s$ shells and $h$ holes; hence shells and holes are all accounted for.

2. The operators above create exactly one vertex. Because all vertices are created, and no vertices created are ever removed, the inversion must include $v - 1$ MEV's.

3. By a similar argument for faces, the inversion must contain $f - s + h$ MEF's.

4. Assuming that there are $m$ MEKR'S, we can use the same argument for rings as well. Hence, the operators already accounted for create $s - h + 1 - m$ rings; since $r$ rings must be created, a total of $r - s + h - 1 + m$ KEMR's are needed.

The account above includes all operators that may appear in an inversion. As a check for the derivation, the reader can verify that these operators create

$$v + f - 2(s - h) - r$$

edges, in accordance with the Euler-Poincaré formula 9.2, page 154. Summing the account, we arrive at the following result:

**Theorem 16.1** *A half-edge data structure with $v$, $e$, $f$, $s$, $h$, and $r$ vertices, edges, faces, shells, holes, and rings can be created with the following counts of Euler operators:*

| | |
|---|---|
| $v - 1$ | MEV*'s* |
| $f - s + h$ | MEF*'s* |
| 1 | MVFS |
| $m$ | MEKR*'s* |
| $r - s + h - 1 + m$ | KEMR*'s* |
| $h$ | KFMRH*'s* |
| $s - 1$ | MFKRH*'s* |

*or a total of*

$$v + f + s + h + r - 2 + 2m \tag{16.1}$$

*operators where $0 \leq m \leq h$. In particular, no operators* KEV, KEF, *and* MVFS *are needed.*

Hence the length of the operator sequence generated by the inversion algorithm grows linearly with the size of the solid.

In practice, some storage is needed for the auxiliary operator RINGMV. Their number can be minimized by removing empty loops immediately as they are created, instead of carrying them along to the face removal phase. With this simple optimization, the storage needed for the operator sequence is in typical cases roughly 30 percent of that of the complete half-edge data structure.

## BIBLIOGRAPHIC NOTES

This chapter is mainly based on the papers [74] and [76]. Hence, odd as it may seem, inversion predates undoing.

Instead of a single undo log, it would be perfectly possible to use a per-solid log. This would make it possible to undo operations on a per-solid basis. Another example of a more involved undo mechanism that also supports a "redo" operation is described in [26]. Here the undo logs are organized according to a CSG tree.

## PROBLEMS

16.1. Write a procedure `printproperties(s)` that calculates the number of components and the genus of each component of the argument solid, $s$.

*Hint:* Modify the procedure `remedg` so that it will not change the genus of the solid. After the modified algorithm has finished, the required outputs can be directly deduced from the data structure.

16.2. Write a procedure `scopy(s)` that creates a copy of its argument solid $s$.

*Hint:* Modify the procedure `invert`.

16.3. How would do undo the copying of a solid performed by the procedure `scopy(s)` of Problem 16.2?

16.4. Consider alternative approaches of undoing the operation `strans(s, m)` that transforms solid $s$ according to the 4-by-4 transformation matrix $m$.

16.5. Describe how the undoing facilities described in this chapter could be used to support the interrupt operation that allows the user to stop a computation and restore the state preceding it. (You may assume that you can specify a procedure that will be asynchronously called if the user hits the interrupt key.)

# Chapter 17

# A USER INTERFACE

The functionality described in the preceding chapters is useful for a user who can develop his/her own applications software on top of them, but not for the ordinary end user who does not wish to write his/her own programs. To satisfy the needs of such a user, we must develop yet a higher layer of GWB, its *user interface*.

## 17.1 LEVELS OF A MODELER REVISITED

One of the underlying themes of Part Two is the recognition of separable levels in solid modeling. The design of GWB is based on a low-level machinery for expressing computations on solid models in the form of Euler operators. On the top of this, higher layers are added, culminating in the advanced algorithms of Chapters 14 and 15.

The modeling concepts visible in each layer are different. In the implementation of Euler operators, considerable care must be put into managing all data items in the half-edge data structure properly. A user that expresses modeling operations in terms of low-level Euler operators must be concerned with providing correct parameters to the operators, and on navigating correctly in the data structure. Finally, solid primitives and set operations emphasize the correct decomposition of a complex part into simpler basic parts, and lower-level details are irrelevant. Clearly, each additional layer is based on additional higher-level, problem oriented modeling primitives.

The same applies for the final layer of the user interface. A successful user interface must be oriented towards the problems that are supposed to be dealt with by the system, and irrelevant details on how things actually take place on lower levels must be effectively hidden. In the analogy

to computer languages suggested in Chapter 8, the roles of the user interface and the so-called "very high-level languages" are similar: in both cases, the user must have natural tools for expressing his/her problem and what he/she wants to be performed, while the details of how this is to be accomplished should be of no concern to him/her.

The design of a good user interface is a demanding task that involves both "hard" technological aspects and "soft" human-oriented aspects. The list of topics that must be taken into account while developing a user interface includes the following:

1. *Conceptual model*: what are the objects the user can manipulate through the user interface?
2. *Functionality*: what operations the user can specify?
3. *Interaction style*: how can the user express his/her wishes by using certain (graphical) input devices?
4. *Context*: how are the inputs entered by the user mapped to the internal objects and operations of the interactive program?
5. *Feedback*: how are the effects of operations and the state of the interactive program represented for the user?
6. *Robustness*: how are errors detected, and recovered from?

A good user interface is based on a clear conceptual model of the application, and all its features are consistent with the model chosen. Like great food, a good user interface is more than just good ingredients put together. Good taste and sense of detail are as important for the developer of a user interface as they are for a chef.

In this text, we shall mainly concentrate on the design of the functionality of the user interface with less regard on the other items. In particular, we shall not deal with the problems of graphical interaction.

One of the assets of the layered modeling system architecture of GWB is that it is possible to develop user interfaces with widely varying modeling concepts. In this chapter, we shall exemplify this by developing two user interfaces to GWB, namely a batch interface based on CSG-style input, and an interactive interface based on "drawing" operations and sweeping.

## 17.2 BATCH INTERFACE

Stripped to its very essentials, a CSG interface involves three families of functions:

```
int  main(argc, argv)
int  argc;
char **argv;
{
    invert(block(atof(argv[2]), atof(argv[3]),
                 atof(argv[4])), argv[1]);
}
```

**Program 17.1** The block primitive program.

1. functions to create primitive solids
2. functions to combine primitives by set operations
3. functions to translate and rotate primitives or combinations of primitives.

By virtue of the inversion algorithm, we can use files in secondary storage to represent the results of each of these groups of functions. In this design, primitive generators accept a few descriptive parameters, and create a file containing the desired object with the inversion algorithm. Set operations and other manipulations read their argument solids from files, and create new files as their results.

Based on the functionality developed in earlier chapters, the implementation of the programs is relatively straightforward. For instance, the program for creating a rectilinear block is implemented with the one-line procedure given in Program 17.1. According to UNIX conventions, the command line is available in the array *argv*; *argc* denotes the number of words in the command line. Here the dimensions of the block are retrieved from the second, third, and fourth argument with help of the UNIX standard procedure `atof` that converts a string of numerals to a floating-point number. The name of the part is taken from the first argument.

In this case, the user interface actually consists of the combination of the UNIX terminal command program, "shell," and programs such as 17.1. The user can consider each such program a "command" that obeys a certain syntax, defined by the way it treats the command line arguments. For instance, assuming that the compiled version of the Program 17.1 is stored in the executable file "`block`," its shell syntax is

```
block <file> <dx> <dy> <dz>
```

where `<file>` is the desired name the of the file for the part, and `<dx>` `<dy>` `<dz>` are the $x$-, $y$-, and $z$-dimensions of the block.

| *Shell syntax* | *Description* |
|---|---|
| `block file dx dy dz [tx ty tz]` | Create block, store result in *file* |
| `cylx file r h [n] [tx ty tz]` | Create cylinder, axis = $x$ |
| `cyly file r h [n] [tx ty tz]` | Create cylinder, axis = $y$ |
| `cylz file r h [n] [tx ty tz]` | Create cylinder, axis = $z$ |
| `conex file r h [n] [tx ty tz]` | Create cone, axis = $x$ |
| `coney file r h [n] [tx ty tz]` | Create cone, axis = $y$ |
| `conez file r h [n] [tx ty tz]` | Create cone, axis = $z$ |
| `torusx file r1 r2 [n1 n2] [tx ty tz]` | Create torus, axis = $x$ |
| `torusy file r1 r2 [n1 n2] [tx ty tz]` | Create torus, axis = $y$ |
| `torusz file r1 r2 [n1 n2] [tx ty tz]` | Create torus, axis = $z$ |
| `ball file r [n] [tx ty tz]` | Create sphere |
| `trans outfile infile tx ty tz` | Translate *infile* by $tx$, $ty$, $tz$ |
| `rotat outfile infile rx ry rz` | Rotate *infile* by $rx$, $ry$, $rz$ |
| `sect file x y z w` | Split *file* with plane $[x\ y\ z\ w]$ |
| `slice outfile infile x y z w` | Calculate cross section |
| `union file1 file2 file3` | Calculate *file1* = *file2* $\cup$ *file3* |
| `inter file1 file2 file3` | Calculate *file1* = *file2* $\cap$ *file3* |
| `minus file1 file2 file3` | Calculate *file1* = *file2* $\setminus$ *file3* |
| `plot [-e ex ey ez] file1 file2 ...` | Draw *file1*, *file2*, ... |
| `plot3 file1 file2 ...` | Draw three views of *file1*, *file2*, ... |
| `volume file1 file2 ...` | Calculate volume of *file1*, *file2*, ... |
| `area file1 file2 ...` | Calculate area of *file1*, *file2*, ... |

**Table 17.1** Batch interface to GWB.

Table 17.1 lists a collection of programs that together implement a batch CSG interface to GWB. All these programs communicate through inversions. In addition to part names and size parameters, this specification includes optional parameters *tx ty tz* that define a translation for the part created. The optional parameter *n* indicates the number of faces used to represent a curved surface.

Observe that cylinders, cones, and tori can be generated in three different default orientations. Hence rigid transformations are needed only in "nonorthogonal" parts. The drawing program **plot** has optional parameters *ex ey ez* that determine the viewing direction. Program **sect** is assumed to store the results of the sectioning into files with some conventional names.

Note that many other operations necessary can be performed by ordinary UNIX tools, such as "`ls`" for listing parts, "`mv`" for renaming parts, and "`rm`" for removing parts.

Program 17.2 gives an example of the batch CSG interface. The "shell program" given creates a simple tray in terms of block and cylinder primitives. The operations in the left column create the "outside" of the tray by subtracting small blocks of material from a larger block, and adding cylinders so as to effect the rounding of its corners. Operations in the right column repeat the same sequence of operations for the "inside" of the tray. The final result is the set difference of the outside and the inside. Of course, by including a scaling program the inside could be created just by scaling the outside.

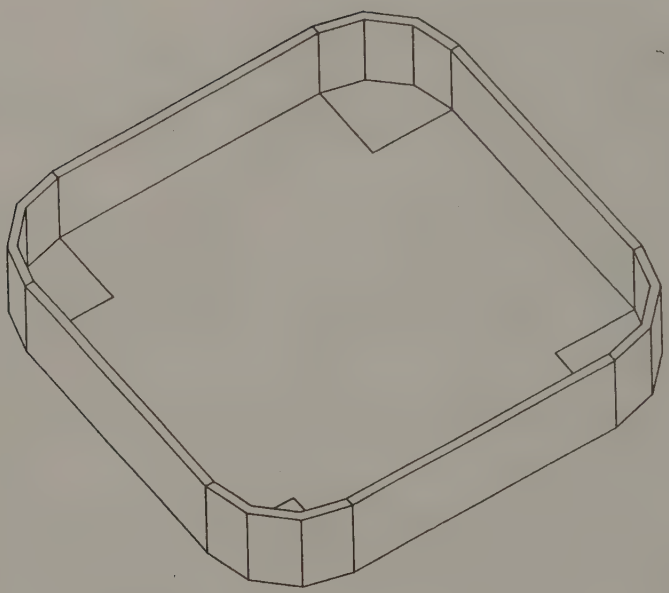
Observe that this description could be written to a file (say, "`tray`"), and evaluated by a single UNIX command

```
sh tray
```

in a batch processing manner. The file can be edited with conventional text editors to alter the design when necessary. Using the parameter substitution facilities of the UNIX shell the design could even be parameterized so as to model a family of trays of various basic dimensions.

## 17.3 INTERACTIVE INTERFACE

The programs of Table 17.1 can be used in a semi-interactive manner by typing commands and drawing intermediate results on a graphical display. However, it is inconvenient and inefficient to add other construction styles besides the CSG input in the batch approach. To include "drafting-type" input, a truly interactive program is needed.



```
block outside 100 100 20
block corn0 20 20 20
minus temp1 outside corn0
block corn1 20 20 20 80 0 0
minus temp2 temp1 corn1
block corn2 20 20 20 80 80 0
minus temp3 temp2 corn2
block corn3 20 20 20 0 80 0
minus temp4 temp3 corn3
cylz add0 20 20 20 20 0
union temp5 temp4 add0
cylz add1 20 20 80 20 0
union temp6 temp5 add1
cylz add2 20 20 80 80 0
union temp7 temp6 add2
cylz add3 20 20 20 80 0
union temp8 temp7 add3
```

```
block inside 96 96 18 2 2 2
block icorn0 20 20 20
minus itemp1 inside icorn0
block icorn1 20 20 20 80 0 0
minus itemp2 itemp1 icorn1
block icorn2 20 20 20 80 80 0
minus itemp3 itemp2 icorn2
block icorn3 20 20 20 0 80 0
minus itemp4 itemp3 icorn3
cylz iadd0 20 18 20 20 2
union itemp5 itemp4 iadd0
cylz iadd1 20 18 80 20 2
union itemp6 itemp5 iadd1
cylz iadd2 20 18 80 80 2
union itemp7 itemp6 iadd2
cylz iadd3 20 18 20 80 2
union itemp8 itemp7 iadd3
minus tray temp8 itemp8
```

**Program 17.2** Example of the batch CSG interface.

### 17.3.1 Interaction Architecture

In analogy to programming languages and compilers, the input of any interactive program can be viewed as a computer language. Just like a programming language must be capable of describing objects (variables, data structures) and operations that can be performed on them (e.g., arithmetic expressions, assignments, and input/output), an interactive geometric program needs facilities for describing geometric objects and for specifying geometric operations on them.

This analogy can be exploited in the actual design of the interactive program. In particular, we can use many of the ideas and techniques developed for compiler construction [2] in the design and implementation of the program.

This approach to the construction of the interactive interface to GWB is summarized in Figure 17.1. The user types sentences of the command language according to the lexical conventions incorporated in a *lexical analyzer*. A *command language parser* recognizes these sentences, and invokes appropriate *command procedures* for performing the task described. All these components of the program use the data available in the half-edge data structure and certain state information to be described in the below.

By virtue of this general architecture, the design of the interactive program can be split into the following tasks:

1. Select a collection of *modeling primitives*, i.e., the basic objects that the user manipulates.

2. Based on the primitives chosen, develop a *command language* for creating primitives, and for performing manipulations on them. This task also includes the selection of the *state variables* and the design of adequate *echoing* of the effects of the commands.

3. Develop a *parser* for the command language, and *command procedures* that actually implement the operations specified.

4. Select the *interaction style*, i.e., the techniques the user can use to express sentences of the command language with the available (graphical) input devices such as a pointing device (mouse), function keys, and ordinary keyboard input. This task is comparable to developing a *lexical analyzer* for the command language parser.

We shall follow this approach, with emphasis on the first three items.

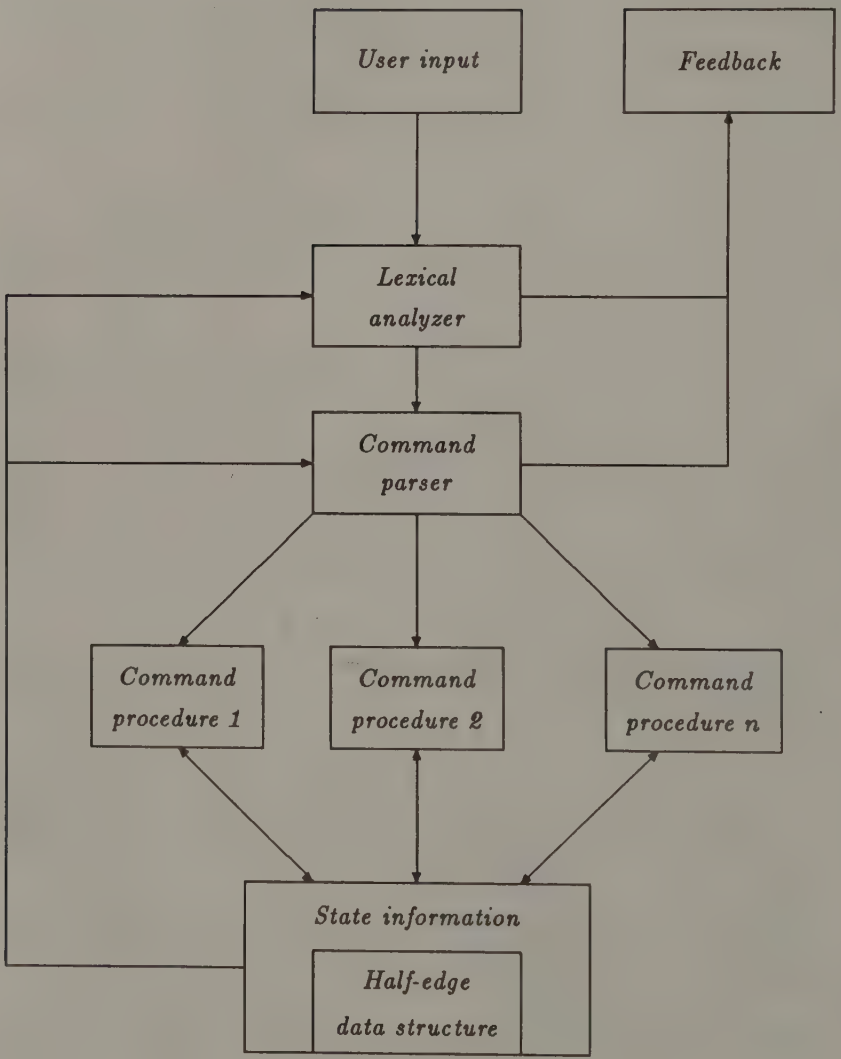



**Figure 17.1** Architecture of the interactive interface.

### 17.3.2 Modeling Primitives

The interactive program is intended to give the user facilities for creating geometric objects by drawing outlines consisting of straight lines and circular arcs, by sweeping outlines, and by combining swept parts in terms of set operations. Clearly, in this case the modeling primitives of interest are

1. points,
2. lines,
3. arcs,
4. outlines, and
5. parts.

The primitives and their respective operations form a hierarchy: outlines consists of lines and arcs, and parts are created from outlines. Similarly, commands for creating and manipulating outlines will be concerned about individual modeling primitives (points, lines, and arcs), while sweeping operations deal with complete outlines, and set operations with complete parts.

### 17.3.3 Anatomy of a Command Language

A command generally involves a *verb*, the part that identifies the operation to be performed, *arguments* that identify the objects processed by the operations, and *parameters* that further describe the operation. For instance, the hypothetical command

```
LINE 100, 150/SOLID
```

consists of the verb LINE, the arguments 100 and 150, and the parameter SOLID. The remaining parts "<space>", ",", and "/" just separate the significant parts from each other and carry no other information.

#### State Information

In addition to explicit arguments and parameters, a command can use implicit state information. For instance, the starting point of the line created by the LINE command above could be implicitly defined to be the "current point." After the execution of the command, the new current point would be the point (100 150).

One of the most important issues in command language design is to decide what information will be included explicitly in a command, and

what information will be derived from the state information in effect at the time when the command is issued, or the *context* for short. A closely related problem to this is the *echoing*: if the interpretation of a command is dependent of the context, it must be indicated to the user. For instance, in the example above the current point could be indicated by a special highlighted cursor symbol.

The hierarchy of modeling primitives and their operations suggests a similar hierarchy in context information. Specifically, a context would consist of the following components:

1. There is always a "current view" (CV) that describes how the parts modeled are visible. The current view consists of a 4-by-4 transformation matrix from part coordinates to window coordinates, and the information defining a windowing operation from the window to the actual display.

2. There is always a "current object" (CO) being under construction. Moreover, there is a linear ordering of all parts created from the most recently created to the least recent part.

3. In the current part, there is always a "current face" (CF) on which the drawing operations take place. CF and all previous current faces have a linear ordering determined by the order they became current.

4. In the current face, there is always a "current point" (CP). Moreover, the points of the current face have a circular ordering determined by their connections to each other. (This ordering often coincides with the order the points were created.)

### Part Commands

After deciding on the state variables, the operations of the command language can be specified by describing their effect on the state information. Hence, after recognizing that a pushdown stack is the natural data structure for recording parts, it is straightforward to specify operations related to parts as follows:

**open** `<name>` Assign name for a part. The next part created will be associated with the name given. If the user does not supply a part name for a part, a unique name is assigned to it by default.

**delete-part** Delete top of the part stack.

**union** Calculate the set union of the two topmost parts of the stack, push result onto the stack.

`minus` As union.

`inter` As union.

`select-part <name>` Make part become the top of the stack.

By virtue of the last operation, the user can modify the ordering of parts if he wants to.

Observe that in the above design, many commands consist of the verb part alone, because argument objects can be derived from the context.

### Drawing Commands

The design of operations for points, lines, and arcs can also exploit the context in a similar fashion:

`move <u> <v>` If no point at $(u\ v)$ exists, create a new point at $(u\ v)$ and make it current. Otherwise, make the point at $(u\ v)$ current.

`line <u> <v>` Draw a line from the current point to a new point at $(u\ v)$, make the new point current. In addition, if the line divides the current face into two faces, select one of them as the new current face.

`arc-cw` Draw a circular arc in clockwise orientation. The arc is centered at the previous point of CP, and delimited by CP and the previous points of the center point. Select the current face as in `line`.

`arc-ccw` Draw a circular arc in counterclockwise orientation. The arc is centered at the previous point of CP, and delimited CP and the previous point of the center point. Select the current face as in `line`.

`next` Advance CP in the circular ordering, i.e. let CP = next point of CP.

`prev` Make new CP = the previous point of CP.

`del` Delete CP, make previous point current.

Observe that points have no names, but are identified indirectly through two coordinates $u$ and $v$. The coordinates $u$ and $v$ are transformed to $x$, $y$, and $z$ based on the current face and information of the current view (CV). Hence, the corresponding point $x$, $y$, and $z$ is calculated by projecting the point $u$, $v$ onto the current face along the viewing direction. This amounts to solving the unknown variables $x$, $y$, $z$ from the three equations determined by the face equation of CF and the requirement that $x$, $y$, $z$ transform to $u$, $v$ under CV.

### Outline Commands

The design is continued by sketching the operations for outlines. Again, a pushdown stack is the natural data structure for recording the state information.

`sweep <depth>` Sweep CF along its normal, make the "top" face of the swept region the new CF by pushing it onto the face stack.

`select-face <u> <v>` Make the nearest face that contains the point $(u\ v)$ current by pushing it onto the face stack.

`prev-face` Make previous current face current again by popping the stack.

`hole` As `prev-face`, but create first a hole by sweeping CF down through the object.

Note that commands for creating lines and arcs can also affect the current face information.

### Viewing Commands

Finally, we need commands that select the current viewing operation:

`eye <ex> <ey> <ez>` Select the viewing direction.

`window <ul> <vb> <ur> <vt>` Select the window.

## 17.4 IMPLEMENTATION

In the UNIX programming environment the implementation of an interactive program can proceed nicely along the lines of Figure 17.1.

### 17.4.1 Lexical Analyzer

The commands outlined in the previous section consist of several types of components, including keywords, coordinates, and part identifiers. The task of the *lexical analyzer* is to recognize these *lexical tokens* from the input stream, and provide them to the parser. In our case, the lexical analyzer becomes a C function `yylex` that returns an integer code denoting the type of the lexical token read, and stores its actual value in a global variable `yylval` as described below. (The obscure names are due to conventions of the parser generator program we use for parser construction.)

Each keyword of the language is considered a distinct lexical token. For the purposes of lexical analysis, keywords and the literals denoting the

```
struct
{
      char        *word;
      int         value;
} w;

struct w keyw[] =
{
      "move",     MOVE,
      "line",     LINE,
      ...
      0,          0,
};
```

**Program 17.3** Keywords of the command language.

token class codes can be represented in an array of records as shown in Program 17.3. Observe that is is easy to define abbreviations of keywords by associating several strings with the identical code.

The recognition of other tokens is based on *lexical conventions* that specify the formats of valid identifiers, numeric constants, and so on. For instance, identifiers are often required to have an alphabetic leading character (`a` ... `z`, `A` ... `Z`), and to consist entirely either of alphabetic characters or of numerals (`0` ... `9`). Under this rule, it is straightforward to distinguish identifiers from numeric constants that must start with a numeral or a unary minus.

Program 17.4 gives an implementation of the lexical analyzer that follows these rules. The analyzer reads user input in terms of the UNIX standard procedure `getchar` that returns the characters typed by the user one at a time. The `while`-loop ignores all newline characters (`"\n"`), spaces, and tabs (`"\t"`); hence these may be inserted in commands at will.

After reading the initial character of a token, the analyzer can immediately decide whether a character string (`strget`) or a numerical value (`numget`) should be read. Keywords are handled by searching a keyword array as defined in Program 17.3; if the token is found, the corresponding numeric value is returned to the parser. Otherwise, the string is assumed to be an identifier, and the code `IDENTIFIER` is returned. The actual string is sent forth by depositing a pointer to the string in the global variable *yylval* available to the parser. Observe that in this design, keywords cannot be used as identifiers.

Numeric values are handled by converting the string to a floating-point

```
extern int yylval;
int c = ' ';

yylex()
{
    int     i;
    int     ret;
    char    str[STRMAX];

    while(c == '\n' || c == ' ' || c == '\t') c = getchar();
    if(c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
    {
        strget(str);
        i = 0;
        while(keyw[i].value)
        {
            if(strcmp(keyw[i].word, str)) return(keyw[i].value);
            i++;
        }
        yylval = str;
        return(IDENTIFIER);
    }
    if(c == '-' || c >= '0' && c <= '9')
    {
        numget(str);
        yylval = atof(str);
        return(NUMBER);
    }
    ret = c;
    c = getchar();
    return(c);
}

strget(str)
char *str;
{
    ...
}

numget(str)
char *str;
{
    ...
}
```

**Program 17.4** Lexical analyzer.

number with the standard UNIX procedure **atof**, and returning the literal NUMBER together with the actual value. All other characters are returned verbatim in terms of their ASCII code interpreted as a number 0...127.

## 17.4.2 Parser and Command Procedures

For the generation of the parser, the UNIX environment includes a very useful software tool YACC (Yet Another Compiler-Compiler) [61].

### YACC—Yet Another Compiler-Compiler

YACC is a software tool for generating a parser from a specification of *grammar rules* and *actions* associated with the rules. The resulting parser may be included as a subroutine into the user's program. The user is also expected to supply a function yylex to act as a lexical analyzer.

The grammar rules are expressed in a style similar to the well-known Backus-Naur form (BNF). For instance, the syntax rule for command window is expressed as follows:

```
window:    WINDOW NUMBER NUMBER NUMBER NUMBER =
           {
                   window($2, $3, $4, $5);
                   newframe();
           }
```

In the above, ":" separates the name of the rule (i.e., a *nonterminal* symbol) from its body, and "=" separates the action from the body. The body consists of five terminal symbols ("WINDOW" and four occurrences of "NUMBER"). The body can also include names of other rules; in particular, it can include its own name to effect recursive syntax definitions. The action is just a C compound statement bounded by "{" and "}". In this case, the action sets the new window definition in effect and redraws the screen.

Terminal symbols are represented by literal values defined in terms of a token statement:

```
%token WINDOW NUMBER
```

By virtue of this definition, "WINDOW" and "NUMBER" become literals denoting some numeric values > 255. These literals can be used in the lexical analyzer in the matter shown in Programs 17.4 and 17.5.

The action part can access the values associated with the lexical tokens or rule names of its body through pseudovariables "$2", "$3", etc. that denote the value associated with the second, third, etc. token or name. The action part can associate a value to its own rule by assigning a value to the pseudovariable "$$".

### YACC Example: Part Commands

Let us now examine the use of YACC with a more elaborate example. For this purpose, we select the part commands.

As noted in the above, a pushdown stack is the natural data structure for parts. Therefore, the first thing needed is the implementation of the stack and its basic routines. This is accomplished in Program 17.5.

"Parts" of this design are simply named solids. Procedure **pushpart** deposits a new part onto the stack; its name is copied with the UNIX library procedure **strcpy**. Procedure **poppart** removes a part from the stack, and returns a pointer to its solid node. Procedure **maketop** arranges a part to become the top of the stack by pushing it onto the stack, and removing the old entry. Procedure **strcmp** is again from the UNIX standard procedure library; it compares two character strings, and returns 1 if they are equal and 0 otherwise.

With these procedures, the part commands can be implemented by the YACC definitions of Program 17.6. In the program, "**%{**" and "**%}**" bound global definitions. Lexical tokens needed for these commands include the token "IDENTIFIER" and a token for each keyword; the actual keywords are specified as in Program 17.3.

By conventions of YACC, "**%%**" separates the foregoing definitions from the actual grammar rules. In this case, the first rule simply states that the language recognized by this parser consists of a list of **stat**'s. The rule for **stat** lists the statements of the language. In both rules, alternative bodies are separated by a "**|**".

The next six rules correspond with the six part commands. The action for rule **openpart** simply records the new part name; the required **pushpart(...)** is performed only after a solid actually is created. The action of the rule **deletepart** removes the topmost part, erases its image (**erasesolid**), and removes the solid (**rmsolid**). The code of **selectpart** makes a part to become the topmost one. Finally, set operations work by fetching the argument parts from the stack, and pushing the result back onto it. The display is updated by erasing the images of the argument solids and drawing the result with the procedure **displaysolid**. The name of the new part is fetched from *curname*; hence the user can give a name for the result in terms of **openpart** before invoking the set operation. Rules for **minus** and **inter** are analogous to **union**.

### Main Program

From input such as that of Program 17.6, YACC creates a procedure called **yyparse**. To construct a working program, a main program that calls it

```
#define       MAXPNAME      30
#define       MAXPSTACK     30

typedef struct
{
      Solid   *psolid;
      char    pname[MAXPNAME];
}     Part;
Part  pstack[MAXPSTACK];
int   pstacktop = -1;

void  pushpart(s, name)
Solid *s;
char  *name;
{
      pstack[++pstacktop].psolid = s;
      strcpy(name, pstack[pstacktop].pname);
}

Solid *poppart()
{
      Solid         *top;
      top = pstack[pstacktop].psolid;
      pstacktop--;
      return(top);
}

void  maketop(name)
char  *name;
{
      for(i = 0; i < pstacktop; i++)
            if(strcmp(name, pstack[i].pname))
            {
                  pushpart(pstack[i].psolid, pstack[i].pname);
                  for(i++; i < pstacktop; i++)
                  {
                        pstack[i-1].psolid = pstack[i].psolid;
                        strcpy(pstack[i].pname, pstack[i-1].pname);
                  }
                  pstacktop--;
                  return;
            }
}
```

**Program 17.5** Part stack operations.

```
%{
char        curname[30];
solid       *s, *sa, *sb, *res;
%}
%token      IDENTIFIER OPENPART DELETEPART
            SELECTPART UNION INTER MINUS
%%
stats:      stat | stats stat;
stat:       openpart | deletepart | selectpart |
            union | minus | inter;
openpart:   OPENPART IDENTIFIER =
            {
                  strcpy($2, curname);
            }
deletepart: DELETEPART =
            {
                  s = poppart();
                  erasesolid(s);
                  rmsolid(s);
            }
selectpart: SELECTPART IDENTIFIER =
            {
                  maketop($2);
            }
union:      UNION =
            {
                  sa = poppart();
                  sb = poppart();
                  erasesolid(sa);
                  erasesolid(sb);
                  res = setop(sa, sb, UNION);
                  displaysolid(res);
                  pushpart(res, curname);
            }
minus:      ...
inter:      ...
```

**Program 17.6** Parser example.

```
int     main(argc, argv)
int     argc;
char    **argv;
{
    int         inputfd;

    if(argc > 1)
    {
        if((inputfd = open(argv[1], 0)) != -1)
        {
            close(0);
            dup(inputfd);
            close(inputfd);
        }
        else fprintf(stderr, "%s: cannot open %s\n",
                    argv[0], argv[1]);
    }
    yyparse();
}
```

**Program 17.7** Main program.

must still be supplied. For the purposes of this text, the code given in Program 17.7 is sufficient.

The procedure makes it possible to use the interactive program also in a batch manner by allowing commands to be read from a file given as an optional argument in the shell command line of the program. This is accomplished by redirecting the terminal input to the file in terms of UNIX system calls `close` and `dup`.

## 17.5 NOTES

In practice, the two user interfaces outlined in this chapter can be tied together by giving the interactive program a facility to write and read inversions, and including the CSG functions into it.

The command language was developed so as to facilitate the use of graphical interaction, when available. Many commands do not take any arguments at all, and can be implemented by a menu selection or function keys. Some others require only two-dimensional position information, and can be implemented by a combination of pointing and menu selection. Hence, the only commands that require typing are the ones involving part

names; they are expected to occur relatively infrequently. Consult any good textbook in computer graphics [88,39] for interactive graphics techniques. See also [77] for one particular approach to the construction of an interactive modeler based on a GWB-like system.

Good tools are indispensable when implementing a nontrivial program. According to the author's experience on the use of parser generators, they can greatly aid the development of virtually any program that has to deal with a collection of commands. When tools such as YACC are available, their use is strongly recommended.

## PROBLEMS

17.1. The user interface described in the text is based on having a collection of "current" entities which are worked on. A more versatile approach to constructing the state information of a "drawing-oriented" user interface is to use a separate *selection buffer* for managing a collection of "selected" graphical entities. The user can edit the buffer, i.e., add or remove items from the buffer, and apply operations to entities stored in the buffer.

Outline an implementation of a selection buffer that can hold any number of half-edge data structure instances. Specify the procedures needed for editing the contents of the buffer.

17.2. Based on the selection buffer of Problem 17.1, outline a command language that supports the similar operations as those described in Section 17.3, but are based on the buffer rather than the "rigid" state information outlined in the text.

17.3. Half-edge data structures are versatile enough to support also drafting-type operations of the kind discussed in Chapter 2. Outline a command language that could be used to draw simple two-dimensional figures consisting of points, lines, and arcs.

*Hint:* Single points could be modeled as a half-edge data structure consisting of a single face and vertex. Single lines could be modeled as one-edge solids. Observe that many of the procedures of Chapter 13 are applicable also for such objects.

# Chapter 18

# EXTENSIONS

The procedures of GWB form a logically complete whole in that they allow solids to be created, processed, and stored in disk files. Nevertheless, to make GWB also practically interesting, enhancements as for functionality and efficiency would be needed. This chapter outlines these extensions to the basic GWB.

## 18.1 EXTENSION OF GEOMETRIC COVERAGE

The geometric coverage of the basic modeler constructed in the second part of this book is not sufficient for tasks where exact modeling of curved surfaces is needed. Hence the extension of the modeler to handle other than merely planar surfaces would be valuable.

Unfortunately, it is difficult to build a solid modeler capable of processing a wide variety of surfaces, and we do not attempt to discuss in detail the construction of a full-scale modeler with curved surfaces. Instead, we describe a much simpler extension that enhances the functionality of the basic modeler while requiring only a modest effort.

Recall that the basic modeler is capable of creating solid objects as parameterized primitives such as blocks, cylinders, and tori. In a sense, the information that a certain object was originally created in this fashion is lost, however: the data structures used have no means of representing that the faces approximating the cylinder surface in Figure 18.1 logically belong together.

Due to this lack of information, the best we can do to render a cylinder is represented in (a), (b), or (c) in the figure. In order to produce images such as (d), (e), the least we need is the capability of telling whether an edge separates two faces arising from distinct surfaces or not.

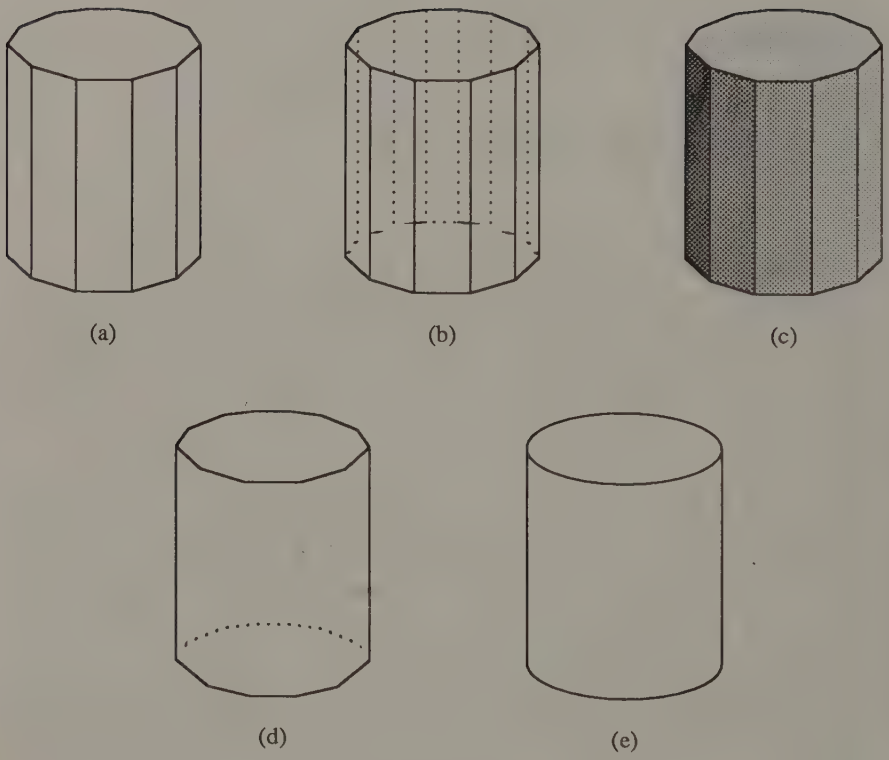



**Figure 18.1** Graphical renderings of a cylinder.

The loss of information causes problems in functionality besides just bad pictures. As we have no concept for distinguishing the set of edges forming an approximation of the intersection curve between the top plane and the cylinder surface, we cannot easily specify operations we might want to perform on this curve as a whole—say, to replace it with a toroidal blending surface.

The solution to these problems is to expand the data structure used in the basic modeler by including concepts for representing the surface information, and by modifying all necessary algorithms to deal properly with this data.

### 18.1.1 Representation of Surfaces

The class of surfaces we are potentially interested in includes all types of surfaces ordinarily found in CSG schemes—that is, natural quadrics and tori. We also want to design the extension so that parametric surfaces can be accommodated, if desired.

One of the leading ideas of boundary representations is the separation of topological and geometrical information. Following this approach, we choose to represent surfaces in "infinite" geometry. For instance, a cylinder surface is represented as an infinite object; the part(s) of it really belonging to object(s) are "chalked out" by edges of the basic modeler. Hence, every surface will be represented by a tuple of parameters that describe it in some convenient orientation, and a transformation matrix that maps it into its true position.

For basic half-spaces, the following parameters could be used:

**Plane:** The four parameters $a$, $b$, $c$, and $d$ of the plane equation

$$[a\ b\ c\ d] \cdot [x\ y\ z\ w]^T = 0$$

need to be stored. Planes are exceptional in that they do not need an explicit transformation matrix; instead, the parameters can be transformed as if they formed the homogeneous coordinates of a point.

**Cylinder:** Cylinders are represented as if their axis were the $z$-axis, and their bottom face on the $xy$-plane. Parameters needed are the radius $r$ and the height $h$ of the cylinder.

**Cone:** Cones are represented similarly as cylinders.

**Sphere:** Spheres are represented as if they were centered at origin. Radius $r$ is the only parameter needed.

**Torus:** Tori are represented as if they were centered at the origin and their axis was along the $z$-axis. Parameters needed are the smaller and larger radii $r_s$, $r_l$.

Program 18.1 outlines the necessary changes in the boundary data structure. The "times-used" counter is necessary for proper memory management.

### 18.1.2 Generation of Surface Information

All procedures of the basic modeler that create faces must be modified so as to deal properly with the new surface information. These include primitives, sweeps, transformations, set operations, splitting, and the undo mechanism.

Most of these modifications are straightforward. Primitives must be extended so as to create surface nodes as required, and assign surface pointers of faces properly. Transformation routines must also transform surfaces. Set operations and splitting must copy the surface information whenever they divide a face into two faces.

Proper changes of the undoing procedures are slightly more tricky. First, undoable procedures for the creation and the deletion of surfaces must be written. For instance, all cylinders would be generated by means of a procedure

```
Cylinder *newcylinder(r, h)
float    r, h;
```

which must be undoable. Hence, the procedure would store the arguments of the corresponding deletion procedure

```
void     delcylinder(c)
Cylinder *c;
```

and vice versa.

Second, the assignment of surface information to polyhedral faces must be made undoable. That is, surface information should be assigned by means of a procedure

```
void     *setsurf(f, s)
Face     *f;
Surface  *s;
```

that must record the previous value of the surface hook into the undo log. The procedure should be written so as to maintain the *times_used* counter;

```
typedef struct
{
        short       surf_type;       /* == PLANE */
        short       times_used;
        real        a, b, c, d;
} Plane;

typedef struct
{
        short       surf_type;       /* == CYLINDER */
        short       times_used;
        real        cy_transf[4][4];
        real        cy_rad;
        real        cy_h;
} Cylinder;

...

typedef union
{
        Plane       p;
        Cylinder    cy;
        Cone        co;
        Sphere      sph;
        Torus       t;
} Surf;

struct face
{
        Id          faceno;          /* face identifier */
        ...
        Surf        *fsurf;          /* surface information */
};
```

**Program 18.1** Data structures for surface representation.

if the counter of a surface drops to zero, the procedure can delete the surface node by means of the appropriate deletion procedure.

Finally, those Euler operators that delete faces (KEF, KFMRH, KVFS) must be modified so as to clear the surface information of the face by means of `setsurf`. In addition, the procedure `applyop` of Program 16.8 must be extended so as to handle the basic operations related to surfaces as well.

After these changes, undoing and all related procedures (such as copying and inversion) will deal properly with surface information.

### Extended Sweeps

Sweeps are the procedures that require the most changes, because we need to mark the faces created by linear sweeping of a sequence of edges approximating a circular arc as belonging to a cylinder surface. But this cannot be done without extending the basic data structure, as it has no concept for circular arcs.

This information can be added in strict analogy to the case of surface information. Hence, each edge will have a pointer to a curve node that includes the sufficient information. A design for straight lines and circular arcs is outlined in Program 18.2. Observe that straight lines do not need explicit information, as their endpoints are available in the boundary data structure through half-edges. As arcs are plane curves, they are represented with the natural parameters and a pointer to the plane they lie on.

With this additional information, the linear sweeping algorithm can examine the curve pointers of edges of the face swept, and tag the faces as planes or cylinders. Similarly, the rotational sweeping algorithm can tag faces generated from straight profile lines as cylinders or cones, and faces generated from arcs spheres or tori.

In order to preserve undoing, curve information should be manipulated by means of procedures similar to the procedures dealing with surface information.

## 18.1.3 Functionality with Surface Information

The addition of surface information into the data structure readily gives the information needed for better image generation. For instance, shading according to the Phong or Gouraud models requires that surface normals are calculated at vertices of a mesh of polygons. Now this information can be formed.

Besides graphics, the ability of grouping related entities of the boundary data structure together has many other uses. The blending of two surfaces was already mentioned in the above. Another wide group of facilities is

```
typedef struct
{
        short       curve_type;      /* == LINE */
        short       times_used;
} Line;

typedef struct
{
        short       curve_type;      /* == ARC */
        short       times_used;
        real        arc_rad;         /* radius */
        real        arc_cx, arc_cy;  /* center */
        real        arc_phi1;        /* start angle */
        real        arc_phi2;        /* end angle */
        plane       *arc_plane;      /* plane of the arc */
} Arc;

...

typedef union
{
        line        l;
        arc         a;
} Curve;

struct edge
{
        HalfEdge    *he1;
        HalfEdge    *he2;
        Curve       *ecurv;          /* curve information */
        ...
};
```

**Program 18.2** Data structures for curve representation.

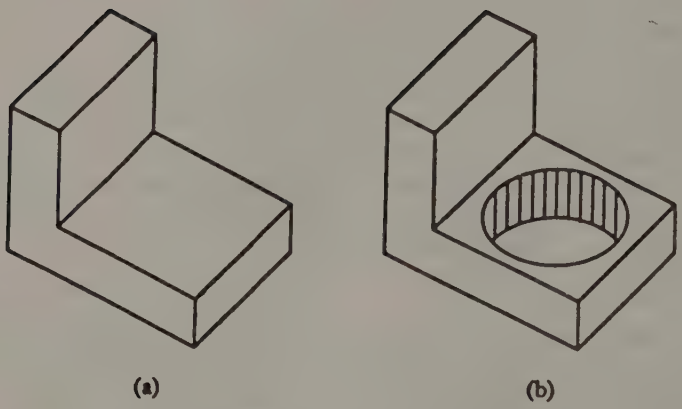



**Figure 18.2** L-brick with a cylindrical hole.

opened by the concept of *feature extraction* which is made possible by the new information. In general, a "feature" is an interrelated group of geometric entities that has significance for some particular problem—say, the set of edges separating two surfaces for the blending problem.

Feature recognition plays a central role in advanced applications of solid modeling. Consider, for instance, the familiar object depicted in Figure 18.2. A program performing planning of machining operations needs the capability of recognizing that a certain set of faces, edges, and vertices forms a "cylindrical hole" in order to decide what kinds of machining operations will modify solid (a) to solid (b).

Model-based *image recognition* gives another case for feature extraction. In this application, features found from a binary image must be related to features extracted from a geometric model, for instance to determine which object is seen and to calculate its orientation.

## 18.2 EFFICIENT GEOMETRIC SEARCH

Solid modeling algorithms are computationally intensive. One of the major challenges of the designer of a solid modeler is to include appropriate mechanisms for speeding up the solid modeling algorithm to an acceptable level.

The general theoretical study of efficient geometric algorithms is the area of *computational geometry*. For good introductions to computational geometry, see [48,110,111,67]. Nevertheless, workers in computational ge-

ometry usually pursue good worst-case asymptotic efficiency by means of advanced data structures and algorithms, while a designer of a solid modeler has much less freedom in the selection of data structures, and must also be concerned with the "practical" expected efficiency of his/her algorithms. Therefore, results of computational geometry are not always applicable to him/her.

This section describes some of the alternative approaches available to the designer of a solid modeler who is concerned on the speed of his/her modeler. For reasons of clarity, we shall consider the prime example of Boolean set operations on solids.

As typical for algorithms of solid modeling, a major factor in the computational cost of set operations is caused by the search of potentially intersecting geometric objects. For instance, the algorithm of Chapter 15 works by comparing all $m$ edges of one solid against the $n$ faces of the other solid. Hence, a total of $mn$ edge-to-face comparisons is involved. While the average cost of per comparison can be made smaller by calculating the enclosing boxes of the two solids and all faces, the $mn$ term will still dominate the asymptotic cost of a set operation.

From the point of view of asymptotic optimality, there is not very much that can be done about this: two polyhedra of sizes $m$ and $n$ can have as many as $O(mn)$ pairs of intersecting faces. In practice, however, cases that really involve a quadratic number of intersections are rare. Much more often the "size" of the overlap is a linear function of $n$ and $m$ rather than quadratic. Observe that two *convex* polyhedra of sizes $n$ have no more than $O(n)$ intersections.

Another typical feature of "practical" cases is that the effect of a set operation is *localized* in the three-dimensional space $E^3$. That is, set operations add or remove material in some fairly restricted part of the space, and leave must of the geometric data untouched. This situation is typical for the interactive use of set operations for design.

A "practical" set operations algorithm should be able to take advantage of these kinds of locality. If the search for intersections can be localized to the area where they are likely to occur, the set operations algorithm should for all practical purposes be capable of working in linear expected time as a function of the size of the inputs.

### 18.2.1 Geometric Queries and Indices

The basic computational step of the set operations algorithm (*"given a query edge (ray), find all faces (boxes) that intersect it"*) typifies the more general problem of providing *access by location* in geometric algorithms. Other examples are listed below:

1. *Box Query*: Given a box, report all geometric entities that intersect it.

2. *Range Query*: Report all geometric entities that are within some distance $r$ from a query location $x, y, z$.

3. *Nearest Neighbors Query*: Report $n$ nearest geometric entities from a query location $x, y, z$.

We call these and identical tasks *geometric queries*. A *geometric index* is defined as a data structure that enhances the efficiency of geometric queries.

Ordinary access methods (B-trees, hashing, etc.) developed for data base management systems are not applicable for geometric tasks, because coordinate values can seldom be utilized as exact keys in access by location. Processing of things with geometric extent is also problematic.

### 18.2.2 Hierarchic Indices

Hierarchic object descriptions belong to the standard techniques of computer graphics for representing complex objects [29]. Indeed, when objects are assembled from transformed instances of other objects as outlined in Section 2.2.2, a hierarchy naturally arises. Octrees of Section 4.2.1 and CSG trees of Section 5.2 are other examples of hierarchic object descriptions we have encountered so far.

The tree structure of a hierarchic object description suggests some obvious mechanisms for speeding up geometric queries. We might include an enclosing box to each node of the tree, i.e., indicate the corners of the smallest rectangular block that completely contains the object represented by the subtree rooted at that node. With this information, the *pruning* of complete subtrees becomes possible. For instance, when processing a ray query, we can test for the intersection of the ray and the enclosing box before looking into a subtree. If the ray does not hit the enclosing box, we can ignore that subtree immediately; see Figure 18.3. Needless to say, this mechanism is ubiquitous in ray casting. It can also be extended to other algorithms, in particular boundary evaluation [119].

Hierarchic indices are most natural when the object description itself is naturally a hierarchy as in the examples listed above. In this case little additional data structures are needed for representing the hierarchic access structure. Nevertheless, hierarchic indices can be applied also when the object description is not hierarchic itself.

In fact, the half-edge data structure described in Chapter 10 includes an elementary hierarchic geometric index. According to Program 10.1, nodes *Solid* and *Face* of the data structure have a "hook" for associating

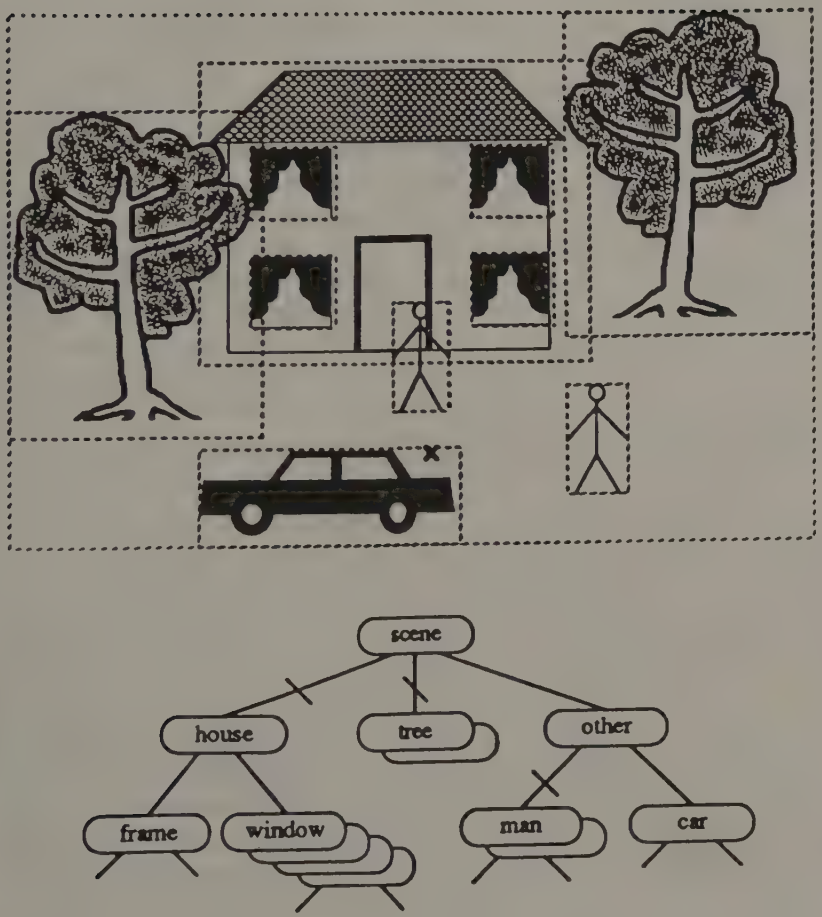



**Figure 18.3** Pruning a hierarchic object description.

an enclosing box with the node. Even this extremely simple mechanism has a profound effect to the speed of algorithms that do lots of geometric search, such as the set operations algorithm.

Recall that the geometric search of the set operations algorithm takes the form of a face-face comparison. Clearly, the information of enclosing boxes can be used to speed up the search of potentially intersecting face pairs. As we have a two-level hierarchy, we can first compare a face against the enclosing box of the other solid; face boxes need to be compared only if this initial test indicates a potential intersection. Usually most potential intersections can be rejected by box intersection tests.

BUILD-2 [18] goes further in this respect, and allows for hierarchies of unlimited depth. In this case, "bubble neighborhoods," i.e., spherical enclosures represented in terms of a center point and a radius, are used instead of rectangular boxes. A special node of the data structure, the "face party," is used to represent the hierarchic index. The parties form a tree rooted at the root node of the data structure having faces at the leaves. Usually faces of a single party have some kind of a logical connection, e.g., they come from the same primitive.

### 18.2.3 Grid Indices

When the object description naturally takes the form of a hierarchy, hierarchic indices are very attractive. However, if this it not the case, there are some drawbacks. In particular, when used for boundary models, the speed gains may deteriorate when the object is modified by algorithms such as set operations and splitting. If the face hierarchy has no correspondence with the spatial distribution of faces, box tests bring little help. Moreover, the rearrangement of the hierarchy so as to make it more useful is itself a complicated geometric problem that definitely requires efficient geometric access.

*Grid indices* form another widely used family of geometric indices that avoids many of the problems of hierarchic indices. They are based on subdividing the space of interest into a set of nonoverlapping *cells*. Each cell is represented by one or several *data pages* that contain (references to) all geometric entities (points, faces) that intersect the cell. A directory mechanism provides for fast access to cells by location.

#### Regular Grid Indices

*Regular grids* are the simplest example of grid indices. In this approach, the space of interest is subdivided into a regular grid of rectangles or boxes

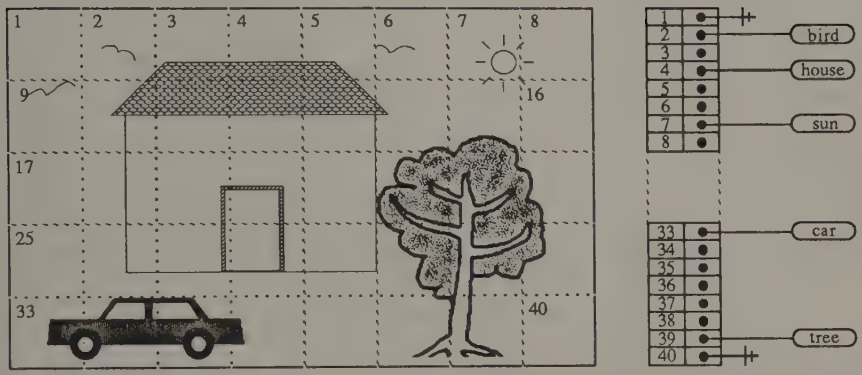



**Figure 18.4** Regular grid index.

that can be thought of as an two- or three-dimensional array of cells. Hence geometric access to cells amounts to simple index calculation.

As all grid indices, regular grids can be used with all kinds of geometric information. For instance, TIPS [90] uses a regular grid as an access mechanism to the half-spaces of a model. A boundary modeler can use a grid index to faces by storing a reference to a face into each cell intersected by the face or its enclosing box.

### Extendible Cell Indices

A drawback of regular grids is that they do not cope very well if the spatial density of the data is highly variable. In the extreme case, most cells would be empty, while the few nonempty cells would need many overflow pages to store all objects that intersect them. Hence, the overall performance deteriorates to that of the exhaustive search.

One way to overcome these difficulties is to let the size of individual cells vary according to the spatial density of data. This approach leads us to an *extendible cell* structure.

**The EXCELL Method** Let us discuss a particular extendible cell mechanism, the EXCELL method [116], in more depth. An EXCELL structure consists of two parts: a *data part* and a *directory*. The cells of the data part are formed by recursively halving the space of interest until the data corresponding with that part of the space fits into a data page. The directory is an array that provides efficient access by location to data pages.

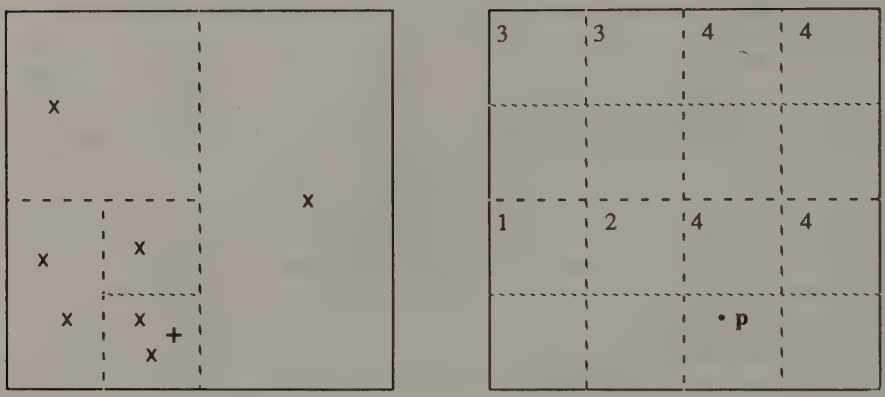



**Figure 18.5** EXCELL concepts.

Figure 18.5 clarifies these concepts. In the figure, the two-dimensional space of interest is divided into four cells (rectangles), none of which contains more than three (= the data page capacity $b$) points. The directory is an array of elements each corresponding to a rectangle of minimal size; each element contains a pointer to the corresponding data page. For instance, the cell containing the query point $p$ is found by calculating the directory array index from the coordinates of $p$, and then fetching the data page pointed at by that array element. The insertion of a new point (+) would require halving of the cell because of overflow; as this cell was of minimal size, the directory elements must be halved also, hence doubling the size of the directory array.

When applied to structured geometric objects (lines, rectangles), EXCELL cells contain pointers to the entities (potentially) intersecting it. Arbitrary objects can be stored by using pointers to enclosing boxes. In these cases, the halving process is controlled by the maximal number of pointers fitting in a data page.

The simple versions of EXCELL described above will fall apart if more than the maximum cell capacity of $b$ objects exactly meet each other. This will happen, for instance, if more than $b$ enclosing boxes intersect each other—an event which is particularly likely if a three-dimensional generalization of the index is used. To overcome this problem, a two-level hierarchy mechanism and overflow capability should be included to the basic index. Figure 18.6 depicts an example of a hierarchic EXCELL structure.

A geometric index should be an easily accessible tool that can be exploited readily in algorithms that need to perform geometric search. In the

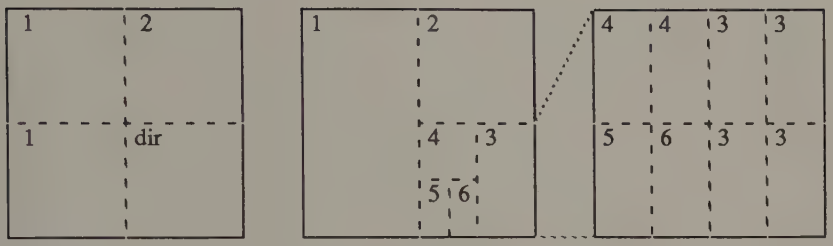



**Figure 18.6** Hierarchic EXCELL.

case of EXCELL, this is possible by selecting a suitable interface between the algorithm and EXCELL.

For maximum flexibility, the index should be capable of storing both points (for storing references to objects having location, but no geometric extent) and boxes (for objects that have geometric extent). The functionality of such a geometric index is represented in the Table 18.1. In particular, the "query object" *cursor* of this design can be a point, a ray, or a box, and the queries supported can include all the types listed above.

**Set Operations With EXCELL** Let us outline how EXCELL could be used to enhance the set operations algorithm of Chapter 15. The obvious way to do this is to replace the exhaustive search for intersections with the edge query supported by EXCELL.

Program 18.3 outlines in C-like pseudocode the implementation of the geometric search for intersections with EXCELL. The code fragment creates an index for one of the argument solids of the set operation, and uses it to find potentially intersecting face pairs. The algorithm assumes that box nodes stored in EXCELL are similar to the box nodes of the half-edge data structure.

If a solid appears as an argument of a sequence of set operations, an index for it may be generated several times. A natural enhancement of the algorithm of Program 18.3 avoids this by constructing the index just once, and by updating it as the final phase of each set operation.

The algorithm of Program 18.4 outlines the index updating operation. The update rules are analogous to the assembly rules for producing the result of the set operation from the boundary classification; for instance, in the case of set union, faces of *ainb* are removed from the geometric index, and faces of *bouta* are inserted to it. Faces split during the set

`index *createindex()`

Create geometric index, return a pointer to it.

`insertindex(root, obj, type)`

Insert object *obj* of type *type* into the index.

`removeindex(root, obj)`

Remove object *obj* from the index.

`deleteindex(root)`

Delete geometric index.

`openquery(root, cursor, type)`

Open geometric query of type *type* with query object *cursor*.

`objtype *nextquery()`

Return the next resulting object of an open query.

**Table 18.1** EXCELL index functions.

```
root = createindex();
fa = sa->sfaces;
while(fa)
{
        if(fa intersects sb->sbox)
                insertindex(root, fa->fbox, BOX);
        fa = fa->nextf;
}
eb = sb->sedges;
while(eb)
{
        if(eb intersects sa->sbox)
        {
                openquery(root, e, EDGE_QUERY)
                while((b = nextquery()) != NIL)
                {
                        process(face of b, eb);
                }
        }
}
```

**Program 18.3** Implementation of the search in set operations.

```
switch(op)
{
        case UNION:
                for(each face f of ainb)
                        removeindex(root, f->fbox);
                for(each face f of bouta)
                        insertindex(root, f->fbox, BOX);
                for(each face f in aoutb adjacent
                    to the intersection polygons)
                {
                        removeindex(root, f->fbox);
                        recalculate f->fbox;
                        insertindex(root, f->fbox, BOX);
                }
        case INTER:
                ...
        case MINUS:
                ...
}
```

**Program 18.4** Index updating in set operations.

operation (which are available by scanning the intersection polygons) must be handled separately. As their geometric extent has changed, a new box should be calculated. In fact, some of them may have no box at all in the index because of the removal of faces in *ainb*.

Experimental results [79] indicate that the enhanced set operations work in approximately linear time in the size of their inputs. Moreover, by virtue of the updating operation, the time needed for repeated set operations on a large model appears to depend only on the size of the "smaller" arguments and the size of the geometric overlap.

In contrast to many theoretical results of computational geometry, calling for very difficult data structures and algorithms, EXCELL is relatively simple to implement. Yet it seems to provide qualitatively the kind of performance enhancement we were originally interested in.

Quite obviously, EXCELL can be used for many other tasks besides the set operations. Ray casting is the most obvious example. In fact, EXCELL can determine the first intersected face (or any other object) along the ray in *constant* expected time. Hence with EXCELL, the computation needed the create a ray-casted image of a scene does not (asymptotically) depend on its complexity but only on the desired resolution.

## BIBLIOGRAPHIC NOTES

The representation and the processing of general curved surfaces is beyond the scope of this text; for information of this field, the reader is referred to some of the many books on the subject [9,38].

After the inclusion of natural quadrics and tori in a modeler, the next logical step would be to offer a blending operation. Recent relevant literature includes [55,84,101,56]. These advances seem to diminish the need for integrating parametric surfaces with solid modelers at least as far blending and fairing is concerned.

While the modeling space of natural quadrics and tori (and their blends) is quite adequate for mechanical parts, many other areas require the capability of processing true free-form parametric surfaces. Unfortunately, the integration of parametric surfaces into a solid modeler is a technically demanding task; for a good overview of the topic, see [40].

The approach followed in GEOMOD [118] is to bite the bullet and represent *all* geometric entities with rational B-splines. In this approach, a major problem is caused by the intersection curves of parametric surfaces, because in general they cannot appear as boundaries of surface patches. Instead, they must be represented approximately in the respective parameter spaces. For instance, the intersection curve of patches $P_1 = P_1(u_1, v_1)$ and $P_2 = P_2(u_2, v_2)$ would be represented by *two* curves $C_1 = C_1(u_1, v_1)$ and $C_2 = C_2(u_2, v_2)$.

Another, quite different approach to the integration of surface patches with solid models is to use an underlying polyhedral model and *smoothen* it by depositing surface patches that correspond with the faces, edges, and vertices of the polyhedron; this, for instance, is the approach chosen in MODIF [25].

Smoothing schemes must somehow deal with the problem of *anomalous regions* that appear at vertices of the original smoothed object. In principle, they would require nonrectangular surface patches. Unfortunately, no generally applicable families of $n$-sided patches, $n \neq 4$, have been presented in the literature. The solid modeler GEOMAP-III [64] avoids this difficulty by automatically subdividing anomalous regions into four-sided regions.

## PROBLEMS

18.1. Based on the surface information of Program 18.1, describe an algorithm that can produce an image such as Figure 18.1(d).

18.2. Specify and implement the procedures for creating and deleting surface nodes and for assigning surface information to faces described in Section 18.1.2.

18.3. Specify and implement the procedures for creating and deleting curves and for assigning curve information to edges.

18.4. Based on the curve information of Program 18.2, outline an extended translational sweeping algorithm that assigns the surface information of the faces generated in the sweep.

18.5. Outline an extended rotational sweeping algorithm that assigns the surface information of the faces generated in the sweep.

18.6. Outline an improved ray casting algorithm for CSG models that makes use of hierarchical enclosures.

18.7. Outline an improved ray casting algorithm for boundary models that makes use of a 3-dimensional regular grid index.

# Appendix A

# HOMOGENEOUS COORDINATES

## A.1 DEFINITION

Many operations of three-dimensional graphics and geometry are most compactly represented in terms of so-called *homogeneous coordinates*. Homogeneous coordinates represent three-dimensional points in terms of four-dimensional ones by adding one additional coordinate axis called $w$. Hence, the three-dimensional point $(x\ y\ z)$ is represented by a four-dimensional point $(x'\ y'\ z'\ w)$ such that

$$\begin{aligned} x &= x'/w \\ y &= y'/w \\ z &= z'/w \end{aligned} \tag{A.1}$$

The homogeneous coordinate representation offers many advantages. For instance, the four-dimensional representation can be scaled at will by multiplying all coordinates by the same factor, as this will not affect the correspondence given by the Equations A.1. Observe also that points at infinity can be conveniently represented by choosing $w = 0$.

## A.2 COORDINATE TRANSFORMATIONS

By far the most important property of homogeneous coordinates is that they allow all *coordinate transformations* needed in viewing and modeling to be represented with 4-by-4 matrices.

### A.2.1 Translation

The *translation transformation* that transforms a point $(x\ y\ z)$ to

$$(x + t_x\ \ y + t_y\ \ z + t_z)$$

is represented by the matrix $T$ constructed as follows:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \tag{A.2}$$

Given the matrix $T$, the translation can be calculated in the homogeneous coordinate representation as the vector-matrix product

$$[x'\ y'\ z'\ w] \times T = [x' + t_x w\ \ y' + t_y w\ \ z' + t_z w\ \ w]^T.$$

The inverse translation is simply

$$T(-t_x, -t_y, -t_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -t_x & -t_y & -t_z & 1 \end{bmatrix} \tag{A.3}$$

### A.2.2 Rotation

The *rotation transformation* that rotates points around one of the coordinate axes has a similar simple structure. The rotation of $\phi$ degrees around the $z$-axis is represented by the matrix $R_z$ as follows:

$$R_z(\phi) = \begin{bmatrix} \cos\phi & \sin\phi & 0 & 0 \\ -\sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.4}$$

Angle $\phi$ represents a counterclockwise rotation; for instance, if $\phi = 90$ degrees, $R_z(\phi)$ maps the point (1 0 0 1) to the point (0 1 0 1). Rotation matrices $R_x$ and $R_y$ that represent corresponding rotations around $x$- and $y$-axes are given by the following similar matrices:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & \sin\phi & 0 \\ 0 & -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.5}$$

$$R_y(\phi) = \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(A.6)}$$

### A.2.3 Scaling

The *scaling transformation* $S(s_x, s_y, s_z)$ can also be expressed as a 4-by-4 matrix:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(A.7)}$$

## A.3 COMBINATION OF TRANSFORMATIONS

A major advantage of homogeneous coordinate transformations is that they can be piled upon each other. Specifically, an arbitrary series of translations and rotations can be combined into a single transformation matrix simply by calculating the product of the corresponding translation or rotation matrices in the correct order. For instance, the composite transformation matrix $M$ consisting of a translation $T = T(t_x, t_y, t_z)$ followed by rotations $R_1 = R_x(\phi_1)$ and $R_2 = R_x(\phi_2)$ can be produced as the matrix product

$$M = T \times R_1 \times R_2.$$

Combined, an arbitrary sequence of rotations will yield a general rotation matrix $R$ of the form

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(A.8)}$$

A useful observation is the fact that the 3-by-3 submatrix $[r_{ij}]$ is *orthogonal*, i.e., its column vectors

$$v_i = [r_{1i}\ r_{2i}\ r_{3i}]^T, i = 1, 2, 3$$

are mutually orthogonal unit vectors. Under $R$, they will be mapped to unit vectors on $x$-, $y$-, and $z$-axes, respectively.

Another useful property of an orthogonal matrix is that its inverse is simply its transpose. Hence the inverse transformation matrix $R^{-1}$ of $R$ is simply

$$R^{-1} = R^T = \begin{bmatrix} r_{11} & r_{21} & r_{31} & 0 \\ r_{12} & r_{22} & r_{32} & 0 \\ r_{13} & r_{23} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.9}$$

## A.4 A VIEWING OPERATION

As a useful example of the homogeneous coordinates, let us consider the problem of performing a *3-dimensional viewing operation*, i.e., generating a two-dimensional projected figure of a three-dimensional object such that could be used to drive a graphical display or a plotter. Again, textbooks on computer graphics should be consulted for the full account of the subject. Here we restrict ourselves to just one particular solution.

### A.4.1 Specification of the Viewing Operation

We specify the viewing operation in terms of two points,

$$eye = (e_x\ e_y\ e_z)$$

and

$$ref = (r_x\ r_y\ r_z)$$

and a vector

$$up = [u_x\ u_y\ u_z].$$

Points *eye* and *ref* constitute the elements of the so-called *synthetic camera analogy* popular in three-dimensional computer graphics. In particular, *eye* gives the location of the camera, and *ref* the location the camera is directed at. Hence the vector

$$dir = eye - ref = [e_x - r_x\ \ e_y - r_y\ \ e_z - r_z]$$

gives the direction vector of the camera from *ref*. For instance, $dir = [0\ 0\ 1]$ corresponds with a camera viewing a "top" view of the scene.

Recall that a vector and a point uniquely define a plane that passes through the point with the normal given by the vector. In our case, *dir* and *ref* define the *viewing plane* onto which we shall project our scene. In the synthetic camera analogy, the viewing plane corresponds with the film plane onto which the image is generated.

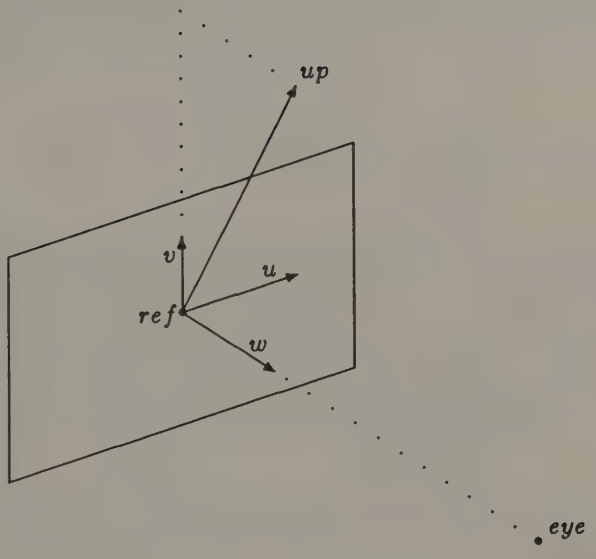



**Figure A.1** Three-dimensional viewing operation.

The coordinates $u$ and $v$ will be used on the viewing plane. To make the viewing operation a coordinate transformation from the $xyz$-space to another three-dimensional space, it is useful to introduce also a third coordinate $w$ that gives "depth" of a point with respect to the viewing plane.[1] To establish the transformation needed, let us fix the origin of the $uvw$-space at the reference point *ref*. Then *dir* establishes the direction vector of $w$-axis in terms of $x$, $y$, and $z$. What remains is the fixing of $u$- and $v$-axes with respect to the $xyz$-space—our synthetic camera points into correct direction, but it may still rotate around the axis perpendicular to the film plane.

The specification is completed with the third piece of input, the vector *up*. Its components $[u_x\ u_y\ u_z]$ give a direction of $xyz$-space that should appear "upwards" (i.e., along the $v$-axis) in the figure. Hence, in a "top" view,

$$[u_x\ u_y\ u_z] = [0\ 1\ 0]$$

makes the $y$-axis to appear upwards in the figure. Likewise,

$$[u_x\ u_y\ u_z] = [0\ 0\ 1]$$

makes the $z$-axis appear upwards. This effect is accomplished by selecting the projection of *up* on the viewing plane as the direction of the $v$-axis. Obviously, *up* must not be perpendicular with the viewing plane.

After the $w$- and $v$-axes have been fixed, the $u$-axis is simply formed by a vector (cross) product of $v$ and $w$. The specification of the viewing operation is illustrated in Figure A.1.

## A.4.2 Construction of the Viewing Matrix

The viewing operation can be compactly represented as a *viewing matrix*, constructed as follows:

1. Create a translation matrix $t$, effecting the change of origin to point *ref*. The resulting $x'y'z'$-space now has the same origin the $uvw$-space.

2. Construct a rotation matrix $r$ that represents the coordinate rotation from the $x'y'z'$-space to the $uvw$-space.

3. Matrix multiply $t$ and $r$ to yield the desired result, matrix *view*.

[1] The depth information is needed for hidden line and hidden surface removal as well.

By using the special properties of orthogonal rotation matrices, both *view* and its inverse $view^{-1}$ can be constructed directly (i.e., without explicit matrix inversion). Let

$$\begin{aligned} [u] &= [u_{x'} \;\; u_{y'} \;\; u_{z'}]^T \\ [v] &= [v_{x'} \;\; v_{y'} \;\; v_{z'}]^T \\ [w] &= [w_{x'} \;\; w_{y'} \;\; w_{z'}]^T \end{aligned}$$

be the unit vectors along the $u$-, $v$-, and $w$-axes in terms of $x'$, $y'$, and $z'$. Then the rotation from $x'y'z'$ to $uvw$ is given by the following matrix:

$$\begin{bmatrix} [u] & [v] & [w] & 0 \\ & 0 & & 1 \end{bmatrix}$$

Because $u$, $v$, and $w$ are orthogonal unit vectors, the inverse transformation is given simply by the transpose matrix:

$$\begin{bmatrix} [\;\; u \;\;]^T & \\ [\;\; v \;\;]^T & 0 \\ [\;\; w \;\;]^T & \\ 0 & 1 \end{bmatrix}$$

Hence the derivation of the viewing matrix reduces to forming the unit vectors along $u$, $v$, and $w$. A procedure that performs this operation is given as Program A.1. It uses the vector and matrix algebra package described in Section A.5.

## A.5 A VECTOR AND MATRIX PACKAGE

This section introduces a collection of simple procedures for processing transformation matrices and doing certain other basic operations for vectors. To simplify the notation, we shall use the C type definitions

```
typedef float vector[4];
typedef float matrix[4][4];
```

```
void    makeview(eye, ref, up, view, viewinv)
vector  eye, ref, up;
matrix  view, viewinv;
{
        matrix          t, r, t1, r1;
        vector          u, v, w;
        double          d;
        int             i;

        w[0] = eye[0] - ref[0];
        w[1] = eye[1] - ref[1];
        w[2] = eye[2] - ref[2];
        normalize(w);
        d = dot(up, w);
        v[0] = up[0] - d*w[0];
        v[1] = up[1] - d*w[1];
        v[2] = up[2] - d*w[2];
        normalize(v);
        cross(u, v, w);
        matident(t);
        mattrans(t, -ref[0], -ref[1], -ref[2]);
        matident(r);
        for(i=0; i<3; i++)
        {
                r[i][0] = u[i];
                r[i][1] = v[i];
                r[i][2] = w[i];
        }
        matmult(view, t, r);
        matident(t1);
        mattrans(t1, ref[0], ref[1], ref[2]);
        matident(r1);
        for(i=0; i<3; i++)
        {
                r1[0][i] = u[i];
                r1[1][i] = v[i];
                r1[2][i] = w[i];
        }
        matmult(viewinv, r1, t1);
}
```

**Program A.1** Construction of the viewing matrix.

that define a 4-vector and a 4-by-4 matrix. These definitions are also used in the main text.

```
matident(m)
matrix m;
```

Make *m* an identity matrix.

```
mattrans(m, tx, ty, tz)
matrix m;
float tx, ty, tz;
```

Combine a translation (*tx ty tz*) with *m* by constructing a translation matrix *t* from *tx*, *ty*, and *tz* and multiplying *m* with *t* on the right.

```
matrotat(m, rx, ry, rz)
matrix m;
float rx, ry, rz;
```

Combine a rotation (*rx ry rz*) with *m* by constructing a rotation matrix *r* from *tx*, *ty* and *tz* and multiplying *m* with *r* on the right. If several rotations are specified, they are applied in the order *rx*, *ry*, *rz*.

```
matmult(m, m1, m2)
matrix m, m1, m2;
```

Calculate the matrix product $m = m1 \times m2$. The procedure should be implemented so that the argument matrix for the result *m* can be same as *m1* or *m2*.

```
vecmult(v1, v2, m)
vector v1, v2;
matrix m;
```

Calculate the vector-matrix product $v1 = v2 \times m$. The procedure should be implemented so that *v1* can equal *v2*.

The following procedures only work on the three first components of their argument vectors.

```
double dot(v1, v2)
vector v1, v2;
```

Calculate the dot (scalar) product of *v1* and *v2*.

```
cross(v1, v2, v3)
vector v1, v2, v3;
```

Calculate the cross (vector) product $v1 = v2 \times v3$. The procedure should be implemented so that the argument vector for the result *v1* can be same as *v2* or *v3*.

```
double normalize(v)
vector v;
```

Make $v$ a unit vector, return its previous length.

```
veccopy(v1, v2)
vector v1, v2;
```

Make *v1* equal to *v2*.

```
vecplus(res, v1, v2)
vector res, v1, v2;
```

Sum vectors *v1* and *v2* into *res*.

```
vecminus(res, v1, v2)
vector res, v1, v2;
```

Let $res = v1 - v2$.

```
vecnull(v, tol)
vector v;
double tol;
```

Test whether $v$ is a null vector up to tolerance *tol*.

```
vecequal(v1, v2)
vector v1, v2;
```

Check whether vectors *v1* and *v2* are the same.

# Appendix B

# ELEMENTS OF POINT SET TOPOLOGY

In the main text we assumed the familiarity with some basic notions of general (point set) topology that can be obtained in undergraduate-level education in mathematics. For self-containedness, the elementary concepts needed are presented in this Appendix, mainly based on the very accessible textbook of Henle [52].

## B.1 CONTINUOUS TRANSFORMATIONS

We shall consider an Euclidean $N$-dimensional space $E^N$. The points of $E^N$ are denoted by $X, Y, Z, \ldots$, and sets of points by $A, B, \ldots$. The distance between two points $X$ and $Y$ is denoted by $d(X, Y)$.[1]

Our first task is to develop a mathematically exact definition for a "continuous transformation." To this end, the most fundamental concept we need is the *neighborhood*:

**Definition B.1** *The* $\delta$-neighborhood *of a point* $X$, $N(X, \delta)$, *is defined as the set of points* $Y$ *such that* $d(X, Y) < \delta$.

In $E^2$, $\delta$-neighborhoods of $X$ are open disks of radius $\delta$ centered at $X$, while in $E^3$ they are open balls.

Intuitively, the condition that a continuous transformation may not cause any splitting or tearing means that it must preserve the nearness of

[1] Topology can be developed without assuming a metric (a distance). Here we have chosen this more easily understandable approach adequate for our practical purposes. Of course, we shall be interested mainly in dimensions $N = 2, 3$.

points and sets: if a point is near some set, also the transformed point must be near the transformed set. To elaborate this intuitive notion, the concept of a neighborhood gives us a precise definition of nearness:

**Definition B.2** *Point $X$ is said to be* near *set $A$, if every neighborhood of $X$ contains a point of $A$.*

Now we can define the fundamental concept of a continuous transformation:

**Definition B.3** *A* continuous transformation *from a set $D$ to a set $R$ is a function $f$ with domain $D$ and range $R$ such that for any point $X \in D$ and set $A \subset D$, if $X$ is near $A$, then $f(X)$ is near the set $f(A) = \{f(Y)|Y \in A\}$.*

We further restrict the class of interesting transformations to *reversible* transformations, i.e., those transformations that have an inverse:

**Definition B.4** *A* topological transformation *is a continuous transformation that has a continuous inverse. Two sets are said to be* topologically equivalent *if there is a topological transformation between them.*

Hence, for instance, the class of all shapes that can be formed from stretching an infinitely elastic ball without tearing or ripping it are topologically equivalent.

## B.2 OTHER TOPOLOGICAL CONCEPTS

The concept of nearness is fundamental in that it forms the basis of defining other topological concepts. Other concepts used in the main text are defined as follows:

**Definition B.5** *A set is* closed *if it contains all its near points. The* closure *of a set $A$, denoted by $c(A)$, is the set of points near $A$.*

**Definition B.6** *A set is* bounded *if it is contained in a neighborhood.*

**Definition B.7** *Sets that are both closed and bounded are said to be* compact.

**Definition B.8** *A set $A$ is* connected *if whenever $A$ is divided into two disjoint, nonempty subsets $B$ and $C$ (i.e., whenever*

$$A = B \cup C,\ B \neq \emptyset,\ C \neq \emptyset,\ B \cap C = \emptyset$$

*holds), one of these sets always contains a point near the other.*

**Definition B.9** *A set A is* open *if no point of A is near the complement of A.*

**Definition B.10** *The* interior *of a set A, denoted by $i(A)$, is the set of points $\in A$ that are not near the complement of A.*

**Definition B.11** *The* boundary *of a set A, $b(A)$, is the set of points that are near both A and its complement.*

## B.3 TOPOLOGICAL SPACES

As seen in the previous section, all topological concepts were built on the notions of nearness and neighborhood. As nearness was defined in terms of neighborhood, we can actually study the topology of any sets by defining the concept of a neighborhood in a suitable way.

This general view of topology is based on the concept of a *topological space*, defined as follows:

**Definition B.12** *A* topological space *is a set $\aleph$ with the choice of a class of subsets of $\aleph$ (each of which is called* neighborhood *of its points) such that each point of $\aleph$ is in some neighborhood, and the intersection of any two neighborhoods of a point contains a neighborhood of that point.*

Based on a particular instance of a topological space, all definitions of the previous sections can be carried out unaltered. For instance, the choice of $\aleph = E^N$ and neighborhoods $= \delta$-neighborhoods satisfies the two conditions above. Hence it leads to a topology on $E^N$. As this choice for a topology of $E^N$ is intuitively the most natural one, it is called the *natural topology* of $E^N$.

It is possible to construct other kinds of topologies by selecting different neighborhoods satisfying the definition of a topological space. The definition of plane models in the main text is based on defining such a *special topology* on $E^2$. In that topology, neighborhoods are not always open disks as usually, but consist on two half-disks on identified edges. The reader is urged to verify that this definition really satisfies the notion of a topological space defined above.

# Bibliography


[1] M. K. Agoston. *Algebraic Topology.* Marcel Dekker, Inc., New York, 1976.

[2] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.

[3] P. Alexandroff. *Elementary Concepts of Topology.* Dover, New York, 1961.

[4] S. Ansaldi, L. De Floriani, and B. Falcidieno. Geometric modeling of solid objects by using a face adjacency graph representation. *Computer Graphics*, 19(3):131–140, July 1985. Proc. SIGGRAPH '85.

[5] A. Appel. Some techniques for shading machine renderings of solids. In *Proc. AFIPS 1968 Spring Joint Computer Conference*, pages 37–45, 1968.

[6] P. Atherton. A scan-line hidden surface removal procedure for constructive solid geometry. *Computer Graphics*, 17(3):73–82, July 1983. Proc. SIGGRAPH '83.

[7] D. Ayala, P. Brunet, and I. Navazo. Object representation by means of nonminimal division of quadtrees and octrees. *Transactions on Graphics*, 4(1):41–59, Jan. 1985.

[8] A. Baer, C. M. Eastman, and M. Henrion. Geometric modeling: a survey. *Computer Aided Design*, 11(5):253–272, 1979.

[9] R. E. Barnhill and W. Böhm, editors. *Surfaces in Computer Aided Geometric Design*, North-Holland, Amsterdam, 1983. Proceedings of a conference held at Mathematisches Forschungsinstitut Oberwolfach, April 25-30, 1982.

[10] A. H. Barr. Superquadrics and angle-preserving transformations. *IEEE Computer Graphics and Applications*, 1(1), 1981.

[11] P. Barr, R. Krimper, M. Lazaer, and C. Stammen. *CAD: Principles and Applications.* Prentice-Hall, Englewood Cliffs, N.J., 1985.

[12] K. Bathe and E. L. Wilson. *Numerical Methods in Finite Element Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[13] B. Baumgart. *Geometric Modelling for Computer Vision.* PhD thesis, Stanford University, 1974. Also as Tech. Rep. CS-463.

[14] B. Baumgart. A polyhedron representation for computer vision. In *National Computer Conference*, pages 589–596, AFIPS Conf. Proc., 1975.

[15] C. B. Besant. *Computer Aided Design and Manufacture.* Ellis Horwood, 1980.

[16] A. Bowyer and J. Woodwark. *A Programmer's Geometry.* Butterworth & Co (publishers) Ltd., Sevenoaks, England, 1983.

[17] J. W. Boyse and J. E. Gilchrist. GMSolid: Interactive modeling for design and analysis of solids. *IEEE Computer Graphics and Applications*, 2(2):86–97, March 1982.

[18] I. C. Braid. *Notes on a Geometric Modeller.* CAD Group Document 101, Computer Laboratory, University of Cambridge, June 1979. Revised 1980.

[19] I. C. Braid, R. C. Hillyard, and I. A. Stroud. Stepwise construction of polyhedra in geometric modeling. In K. W. Brodlie, editor, *Mathematical Methods in Computer Graphics and Design*, pages 123–141, Academic Press, London, 1980.

[20] F. P. Brooks. *The Mythical Man-Month.* Addison-Wesley, Reading, Mass., 1975.

[21] C. M. Brown. PADL-2: A technical summary. *IEEE Computer Graphics and Applications*, 2(2):69–84, March 1982.

[22] I. Carlbom, I. Chakravarty, and D. Vanderschel. A hierarchical data structure for representing the spatial decomposition of 3D objects. *IEEE Computer Graphics and Applications*, 5(4):24–31, Apr. 1985.

[23] M. S. Casale and E. L. Stanton. An overview of analytic solid modeling. *IEEE Computer Graphics and Applications*, 5(2):45–56, Feb. 1985.

[24] S. H. Chasen. *Geometric Principles and Procedures for Computer Graphic Applications*. Prentice-Hall, Englewood Cliffs, N.J., 1978.

[25] H. Chiyokura and F. Kimura. Design of solids with free-form surfaces. *Computer Graphics*, 17(3):289–298, July 1983. Proc. SIGGRAPH '83.

[26] H. Chiyokura and F. Kimura. A method of representing the solid design process. *IEEE Computer Graphics and Applications*, 5(4):32–41, Apr. 1985.

[27] H. Chiyokura and F. Kimura. A representation of solid design process using basic operations. In T. L. Kunii, editor, *Proc. Computer Graphics Tokyo '84*, Springer Verlag, Berlin, Apr. 1984.

[28] A. H. J. Christessen. Approximation of a donut. *Computer Graphics*, 14(3), July 1980. Front page picture of Proc. SIGGRAPH '80.

[29] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[30] H. M. S. Coxeter. *Regular Polytopes*. Dover, New York, 1961.

[31] L. J. Doctor and J. G. Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics and Applications*, 1(3):29–38, July 1981.

[32] R. M. Dunn and B. Herzog, editors. *Computer-Aided Design, Engineering, and Drafting*. Auerbach Publishers Inc., New Jersey, 1984.

[33] C. M. Eastman and M. Henrion. GLIDE: A language for design information systems. *Computer Graphics*, 11(2):24–33, 1977. Proc. SIGGRAPH '77.

[34] C. M. Eastman and K. Weiler. Geometric modeling using the Euler operators. In *Proc. First Annual Conf. on Computer Graphics in CAD/CAM Systems*, pages 248–254, MIT, Apr. 1979.

[35] W. S. Elliott. Interactive graphical CAD in mechanical engineering design. *Computer Aided Design*, 10(2), 1978.

[36] J. Encarnação, editor. *Computer Aided Design—Modeling, Systems Engineering, CAD Systems.* Springer Verlag, Berlin, 1980. Lecture Notes in Computer Science No. 89.

[37] J. Encarnação and E. G. Schlechtendahl, editors. *Computer-Aided Design Fundamentals and System Architecture.* Springer Verlag, New York, 1983.

[38] I. D. Faux and M. J. Pratt. *Computational Geometry for Design and Manufacture.* John Wiley, New York, 1979.

[39] J. D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Mass., 1982.

[40] R. Forrest. Freeform surfaces and solid modelling. In *SIGGRAPH '84 Course Notes on Freeform Curves and Surfaces*, 1984.

[41] A. Fournier, D. Fussell, and L. C. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.

[42] I. Gargantini. Linear octrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20:365–374, 1982.

[43] D. L. Goetsch. *Computer-Aided Drafting.* Prentice-Hall, Englewood Cliffs, N.J., 1985.

[44] R. Goldstein. Defining the bounding edges of a SynthaVision solid model. In *Proc. 18th Design Automation Conference*, ACM/IEEE, 1981.

[45] R. Goldstein and L. Malin. 3D modeling with the SynthaVision system. In *Proc. First Annual Conference on Computer Graphics in CAD/CAM Systems*, pages 244–247, Apr. 1979.

[46] R. A. Goldstein and R. Nagel. 3-D visual simulation. *Simulation*, 16(1):25–31, Jan. 1971.

[47] R. A. Guedj, editor. *Methodology in Computer Graphics*, North-Holland, Amsterdam, 1978. Proceedings of the IFIP WG 5.2 workshop held in Seillac, France, July 1986.

[48] L. J. Guibas and J. Stolfi. Notes on computational geometry. 1983. Lecture Notes, Stanford University.

[49] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computations of Voronoi diagrams. *Transactions on Graphics*, 4(2):74–123, Apr. 1985.

[50] P. Hanrahan. Creating volume models from edge-vertex graphs. *Computer Graphics*, 16(3):77–84, July 1982. Proc. SIGGRAPH '82.

[51] F. Harary. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969.

[52] M. Henle. *A Combinatorial Introduction to Topology.* W. H. Freeman and Company, San Francisco, 1979.

[53] R. Hillyard. The BUILD family of solid modelers. *IEEE Computer Graphics and Applications*, 2(2), March 1982.

[54] E. Hinton and D. R. J. Owen. *Finite Element Programming.* Academic Press, London, 1977.

[55] C. Hoffman and J. Hopcroft. Automatic surface generation in computer aided design. *The Visual Computer*, 1:92–100, 1985.

[56] L. Holmström. *Piecewise Quadric Blending of Implicitly Defined Surfaces.* Report TKK-TKO-B60, Helsinki University of Technology, Espoo, 1986. 24 p.

[57] F. R. A. Hopgood, D. A. Duce, J. R. Gallop, and D. C. Sutcliffe. *Introduction to the Graphical Kernel System (GKS).* Academic Press, New York, 1983.

[58] W. A. Hunt. *An Explatoratory Study of Automatic Verification of Programs for Numerically Controlled Machine Tools.* Master's thesis, Mechanical and Aerospace Sciences Department, University of Rochester, 1981.

[59] G. M. Hunter. *Efficient Computation and Data Structures for Graphics.* PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N. J., 1978.

[60] C. L. Jackins and S. L. Tanimoto. Octtrees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14:249–270, 1980.

[61] S. C. Johnson. *YACC—Yet Another Compiler-Compiler.* Bell Laboratories, Murray Hill, N.J., 1978.

[62] J. T. Kajiya. New techniques for ray tracing procedurally defined objects. *ACM Transactions on Graphics*, 2(3):161–181, July 1983.

[63] E. Kawaguchi and T. Endo. On a method of binary picture representation and its application to picture compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):27–35, 1980.

[64] F. Kimura. Geomap-III: Designing solids with free-form surfaces. *IEEE Computer Graphics and Applications*, 4(6):58–72, June 1984.

[65] P. Koistinen, M. Tamminen, and H. Samet. Viewing solid models by bintree conversion. In *Proc. EUROGRAPHICS '85*, pages 147–157, North-Holland Publ. Co., Amsterdam, 1985.

[66] T. L. Kunii, T. Satoh, and K. Yamaguchi. Generation of topological boundary information from octree encoding. *IEEE Computer Graphics and Applications*, 5(3):29–38, March 1985.

[67] D. T. Lee and F. P. Preparata. Computational geometry—a survey. *IEEE Transactions on Computers*, C-33(12):1072–1101, Dec. 1984.

[68] Y. T. Lee and A. A. G. Requicha. Algorithms for computing the volume and other integral properties of solid objects. II. a family of algorithms based on representation conversion and cellular approximation. *Communications of the ACM*, 25(9):642–650, Sep. 1982.

[69] Y. T. Lee and A. A. G. Requicha. Algorithms for computing the volume and other integral properties of solid objects. I. known methods and open issues. *Communications of the ACM*, 25(9):635–641, Sep. 1982.

[70] J. A. Levin. A parametric algorithm for drawing pictures of solid objects composed of quadric surfaces. *Communications of the ACM*, 19(10):642–650, 1976.

[71] V. C. Lin, D. C. Gossard, and R. A. Light. Variational geometry in computer-aided design. *Computer Graphics*, 15(3):171–177, Aug. 1981. Proc. SIGGRAPH '81.

[72] B. B. Mandelbrot. *Fractal Geometry of Nature*. W. H. Freeman, San Francisco, 1983.

[73] M. Mäntylä. Boolean set operations of 2-manifolds through vertex neighborhood classification. *Transactions on Graphics*, 5(1):1–29, Jan. 1986.

[74] M. Mäntylä. An inversion algorithm for geometric models. *Computer Graphics*, 16(3):51–59, 1982. Proc. SIGGRAPH '82.

[75] M. Mäntylä. A note on the modeling space of Euler operators. *Computer Vision, Graphics, and Image Processing*, 26(1):45–60, 1984.

[76] M. Mäntylä. Undo support in HutDesign. In *Proc. CAPE '86.*, North-Holland Publ. Co., Amsterdam, 1986.

[77] M. Mäntylä and M. Ranta. Interactive solid modeling in HutDesign. In T. L. Kunii, editor, *Proc. Computer Graphics Tokyo '86*, Springer Verlag, Tokyo, 1986.

[78] M. Mäntylä and R. Sulonen. GWB—a solid modeler with Euler operators. *IEEE Computer Graphics & Applications*, 2(7):17–31, 1982.

[79] M. Mäntylä and M. Tamminen. Localized set operations for solid modeling. *Computer Graphics*, 17(3):279–288, 1983. Proc. SIGGRAPH '83.

[80] G. Markowsky and M. A. Wesley. Fleshing out wire frames. *IBM Journal of Research and Development*, 24(5):582–597, 1980.

[81] J. Mayer. *Algebraic Topology.* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[82] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.

[83] D. Meagher. *Octree Encoding: A New Technique for the Representation, Manipulation, and Display of Arbitrary Three-Dimensional Objects by Computer.* Technical Report IPL-TR-80,111, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, New York, Oct. 1980.

[84] A. E. Middleditch and K. H. Sears. Blend surfaces for set theoretic volume modelling systems. *Computer Graphics*, 19:161–170, 1985. Proc. SIGGRAPH '85.

[85] M. Mortenson. *Geometric Modeling.* John Wiley & Sons, New York, 1985.

[86] I. Navazo, D. Ayala, and P. Brunet. A geometric modeller based on the exact octree representation of polyhedra. *Computer Graphics Forum*, 5(2):89–104, June 1986.

[87] R. G. Newell. Solid modelling and parametric design in the Medusa system. In *Computer Graphics 82*, pages 223–235, 1982. Proceedings of the Online Conference.

[88] W. E. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, New York, N.Y., 2nd edition, 1979.

[89] W. N. Newman and A. van Dam. A brief history of efforts towards graphics standardization. *ACM Computing Surveys*, 10(4):365–380, Dec. 1978.

[90] N. Okino, Y. Kakazu, and H. Kubo. TIPS-1: technical information processing system for computer-aided design, drawing and manufacturing. In J. Hatvany, editor, *Computer Languages for Numerical Control*, page 141, North-Holland, Amsterdam, 1973.

[91] N. Okino et al. *Technical Information Processing System TIPS-1.* Institute of Precision Engineering, Hokkaido University, 1978.

[92] T. Pavlidis. *Algorithms for Graphics and Image Processing.* Computer Science Press, 1981.

[93] D. R. Reddy and S. Rubin. *Representation of Three-Dimensional Objects.* CMU-CS-78-113, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Apr. 1978.

[94] A. A. G. Requicha. *Mathematical Models of Rigid Solids.* Tech. Memo. No. 28, Production Automation Project, University of Rochester, 1977.

[95] A. A. G. Requicha. Representations of solid objects—theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, Dec. 1980.

[96] A. A. G. Requicha and H. B. Voelcker. Boolean operations in solid modelling: Boundary evaluation and merging algorithms. *Proc. IEEE*, 3(7):30–44, Oct. 1983.

[97] A. A. G. Requicha and H. B. Voelcker. *Constructive Solid Geometry.* Tech. Memo. No. 25, Production Automation Project, University of Rochester, 1977.

[98] A. A. G. Requicha and H. B. Voelcker. An introduction to geometric modeling and its applications in mechanical design and production. In J. Tou, editor, *Advances in Information Systems Sciences*, Plenum Publishing Co., 1981.

[99] A. A. G. Requicha and H. B. Voelcker. Solid modeling: a historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2(2):9–24, March 1982.

[100] A. A. G. Requicha and H. B. Voelcker. Solid modeling: current status and research directions. *IEEE Computer Graphics and Applications*, 3(10):25–37, Oct. 1983.

[101] A. Rockwood and J. Owen. Blending surfaces in solid modeling. 1985. To be published.

[102] D. F. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill, New York, N.Y., 1985.

[103] D. F. Rogers and J. A. Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, New York, N.Y., 1976.

[104] S. D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, 1982.

[105] M. A. Sabin. Geometric modelling — Fundamentals. In P. J. W. ten Hagen, editor, *Eurographics Tutorials '83*, chapter 9, pages 343–390, Springer Verlag, Berlin, 1984.

[106] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16:187–260, 1984.

[107] H. Samet and M. Tamminen. Bintrees, CSG trees and time. *Computer Graphics*, 19(3):121–130, 1985. Proc. SIGGRAPH '85.

[108] H. Samet and M. Tamminen. Computing geometric properties of images represented by linear quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(2):229–240, 1985. Also appeared in Computer Science Department report series, University of Maryland, Report TR-1359, 1983.

[109] R. F. Sarraga. Algebraic methods for intersections of quadric surfaces in GMSolid. *Computer Graphics and Image Processing*, 22:222–238, 1983.

[110] M. I. Shamos. *Computational Geometry*. PhD thesis, Yale University, 1977.

[111] M. I. Shamos and F. Preparata. *Computational Geometry*. Springer Verlag, New York, 1985.

[112] S. N. Shirari. Representation of three-dimensional digital images. *ACM Computing Surveys*, 13(4):399–423, 1981.

[113] I. Sommerville. *Software Engineering*. Addison-Wesley, Reading, Mass., 1985.

[114] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974.

[115] M. Tamminen. Encoding pixel trees. *Computer Vision, Graphics and Image Processing*, 28:44–57, 1984.

[116] M. Tamminen. *The EXCELL Method for Efficient Geometric Access to Data.* Volume 34 of *Acta Polytechnica Scandinavica, Mathematics and Computer Science Series*, Finnish Academy of Technical Sciences, Helsinki, 1981. 57 p.

[117] M. Tamminen and F. Jansen. An integrity filter for recursive subdivision meshes. *Computers & Graphics*, 9(4):351–363, 1985.

[118] W. Tiller. Rational B-splines for curve and surface representation. *IEEE Computer Graphics and Applications*, 3(6), Sep. 1983.

[119] R. B. Tilove. *Exploiting Spatial and Structural Locality in Geometric Modeling.* PhD thesis, University of Rochester, 1981. Available as Tech. Memo. No. TM-38, Production Automation Project.

[120] R. B. Tilove. A null-object detection algorithm for constructive solid geometry. *Communications of the ACM*, 27(7):684–694, July 1984.

[121] R. B. Tilove. Set membership classification: a unified approach to geometric intersection problems. *IEEE Transactions on Computers*, C-29(10):847–883, 1980.

[122] H. G. Timmer and J. M. Stern. Computation of global geometric properties of solid objects. *Computer Aided Design*, 11(6), Nov. 1980.

[123] J. J. Udupa. Display of 3D information in discrete 3D scenes produced by computerized tomography. *Proc. IEEE*, 71(3):420–433, March 1983.

[124] J. J. van Wijk. Ray tracing objects defined by sweeping a sphere. In *Proc. Eurographics '84*, pages 73–82, Copenhagen, Sep. 1984. Reprinted in Computers and Graphics, Vol 9. No 3, 1985.

[125] J. J. van Wijk. Ray tracing objects defined by sweeping planar cubic splines. *ACM Transactions on Graphics*, 3(3):223–237, July 1984.

[126] H. B. Voelcker and A. A. G. Requicha. Geometric modeling of physical parts and processes. *IEEE Computer*, 10(2):48–57, 1977.

[127] H. B. Voelcker. et al. The PADL-1.0/2 system for defining and displaying solid objects. *Computer Graphics*, 12(3):257, July 1978. Proc. SIGGRAPH '78.

[128] D. L. Vossler. Sweep-to-CSG conversion using pattern recognition techniques. *IEEE Computer Graphics and Applications*, 5(8):61–68, Aug. 1985.

[129] M. A. Wesley. Construction and use of geometric modeling systems. In J. Encarnação, editor, *Computer Aided Design—Modeling, Systems Engineering, CAD Systems*, Springer Verlag, Berlin, 1980.

[130] T. Woo. Feature extraction by volume decomposition. In *Proc. Conference on CAD/CAM Technology in Mechanical Engineering*, pages 76–94, Cambridge, Mass., Mar. 24–26, 1982.

[131] J. R. Woodwark. *Computing Shape.* Butterworth & Co (publishers) Ltd., Sevenoaks, England, 1986.

[132] J. R. Woodwark. Generating wireframes from set-theoretic solid models by spatial division. *Computer Aided Design*, 18(6):307–315, July 1986.

[133] J. R. Woodwark and K. M. Quinlan. The derivation of graphics from volume models by recursive subdivision of the object space. In *Proc. CAD '80*, Online Publications, Northwood Hills, 1980.

[134] F. Yamaguchi and T. Tokieda. Bridge edge and triangulation approach in solid modeling. In T. L. Kunii, editor, *Proc. Computer Graphics Tokyo '84*, Springer Verlag, Berlin, Apr. 1984.

[135] F. Yamaguchi and T. Tokieda. A unified algorithm for Boolean shape operations. *IEEE Computer Graphics and Applications*, 4(6):24–37, June 1984.

[136] K. Yamaguchi, T. L. Kunii, K. Fujimura, and H. Toriya. Octree-related data structures and algorithms. *IEEE Computer Graphics and Applications*, 4(1):53–59, 1984.

[137] M. M. Yau and S. N. Shirari. A hierarchical data structure for multidimensional images. *Communications of the ACM*, 26(7):504–515, 1983.

# List of Programs


| | | |
|---|---|---|
| 2.1 | Type *point* | 14 |
| 2.2 | Type *line* | 15 |
| 2.3 | Another definition of *point* and *line* | 16 |
| 2.4 | Type *object* | 18 |
| 2.5 | Definition of a box | 20 |
| 2.6 | Pyramid | 21 |
| 2.7 | Points with constraints | 24 |
| 4.1 | Octree data structures | 65 |
| 4.2 | Construction of an octree | 66 |
| 5.1 | Divide and conquer for CSG trees | 86 |
| 5.2 | Edge-solid classification | 87 |
| 5.3 | Wire frame generation algorithm | 90 |
| 5.4 | Ray classification algorithm | 94 |
| 10.1 | C definition of the half-edge data structure | 168 |
| 10.1 | C definition of the half-edge data structure (cont.) | 169 |
| 10.2 | Other definitions | 170 |
| 10.3 | Hierarchic access | 172 |
| 10.4 | Neighbor halfedge access | 173 |
| 11.1 | Storage allocation | 176 |
| 11.2 | Linked list manipulation | 178 |
| 11.3 | Halfedge manipulation | 180 |
| 11.4 | Halfedge deletion | 181 |
| 11.5 | The `mvfs` operator | 182 |
| 11.6 | The `lmev` operator | 184 |
| 11.7 | The `lmef` operator | 185 |
| 11.8 | The `lkemr` operator | 187 |
| 11.9 | Scanning procedures | 188 |
| 11.10 | The `mev` operator | 189 |
| 12.1 | Arc generator | 201 |
| 12.2 | Circle generator | 202 |
| 12.3 | Translational sweeping | 203 |

12.4 Block and cylinder primitives . . . . . 204
12.5 Rotational sweeping . . . . . 207
12.6 Ball primitive . . . . . 208
12.7 The gluing procedure . . . . . 209
12.8 Joining of solids . . . . . 210
12.9 The loop gluing procedure . . . . . 212
12.10 Torus primitive . . . . . 213
12.11 Rotational sweeping of closed figures . . . . . 215
13.1 Evaluation of face equations . . . . . 219
13.2 Vertex equality . . . . . 220
13.3 Vertex-edge intersection . . . . . 222
13.4 Boundary cases . . . . . 223
13.5 Vertex-loop containment . . . . . 225
13.6 Line intersection . . . . . 227
13.7 Volume evaluation . . . . . 228
13.8 Loop area evaluation . . . . . 230
14.1 Outline of the splitting algorithm . . . . . 241
14.2 Reduction step . . . . . 243
14.3 Vertex neighborhood classifier . . . . . 246
14.4 Initial classification . . . . . 247
14.5 Reclassification of ON-sectors . . . . . 249
14.6 Reclassification of ON-edges . . . . . 250
14.7 Insertion of null edges . . . . . 252
14.8 Neighborship test . . . . . 252
14.9 The joining algorithm . . . . . 256
14.10 Joining and cutting of null edges . . . . . 257
14.11 Generation of results . . . . . 259
14.12 Classification of faces . . . . . 260
15.1 Outline of the set operations algorithm . . . . . 272
15.2 Generation of relevant vertices . . . . . 274
15.3 Edge-face comparison . . . . . 275
15.4 Vertex-face comparison . . . . . 276
15.5 Vertex neighborhood classification . . . . . 277
15.6 Vertex-vertex neighborhood classifier . . . . . 279
15.7 Search for intersecting sectors . . . . . 281
15.8 Preprocessing of sectors . . . . . 282
15.9 Sector intersection test . . . . . 284
15.10 Reclassification of "on"-sectors . . . . . 287
15.10 Reclassification of "on"-sectors (cont.) . . . . . 288
15.11 Insertion of null edges . . . . . 292
15.12 Auxiliaries for the insertion of null edges . . . . . 293
15.13 Joinability test for set operations . . . . . 295

15.14 Joining algorithm for set operations . . . . . 296
15.15 Generation of the result . . . . . 297
16.1 Undo log data structures and operations . . . . . 305
16.2 The undoable `lmev` operator . . . . . 306
16.3 The undoable `lkef` operator . . . . . 308
16.4 Division of a solid . . . . . 309
16.5 Undoable division of a solid . . . . . 310
16.6 Transaction log management . . . . . 311
16.7 Undoing single operations . . . . . 312
16.8 Applying undo log records . . . . . 314
16.9 Outline of the inversion algorithm . . . . . 318
16.10 Removal of edges . . . . . 319
16.11 Removal of faces . . . . . 321
16.12 Reading an inversion . . . . . 322
17.1 The block primitive program . . . . . 327
17.2 Example of the batch CSG interface . . . . . 330
17.3 Keywords of the command language . . . . . 337
17.4 Lexical analyzer . . . . . 338
17.5 Part stack operations . . . . . 341
17.6 Parser example . . . . . 342
17.7 Main program . . . . . 343
18.1 Data structures for surface representation . . . . . 349
18.2 Data structures for curve representation . . . . . 351
18.3 Implementation of the search in set operations . . . . . 361
18.4 Index updating in set operations . . . . . 362
A.1 Construction of the viewing matrix . . . . . 372

# Index


2-manifold 35–37
    definition of 36
    orientability of 43
    manipulation of 139

Adaptive subdivision 63, 357
addeulerop 305
addhe 180
addlist 178, 259
addsoov 243
addsovf 276
addsovv 276
Analytic modeler 128, 136
Ansaldi, S. 174
Appel, A. 98
applyop 314
    surface information 350
arc 201
Atherton, P. 128
Ayala, P. 129

Baer, A. 120
ball 208
Baumgart, B. 106, 147
BeginTransaction 311
Betti numbers 45, 48, 140
Bintree 72, 98
bisector 247, 262
Blending 115, 125, 347, 363
block 204, 327
bndrlv 223
Boundary classification 267, 269
Boundary evaluation 91, 124
    incremental 91, 126
    inverse 124
    using hierarchical enclosures
        for 354
Boundary representation 101–121
    conversion to an octree of 130
    in a hybrid modeler 126
Bowyer, A. 26
Braid, I. C. 120, 156, 160
B-splines 363
BUILD-2 114, 115, 121, 216, 356

canjoin 256
Characteristic function 77
Chasen, S. H. 26
checkwideness 231, 247
circle 202
Classification theorem 143
classify 260
cleanup 262
comp 220, 242
Completeness 6
    of Euler operators 159
    of plane model operations 145
Computational geometry 352
Connected minus operation 144
Connected sum operation 143, 151
Connectivity 45, 144, 259
    evaluation of 324
    of exhaustive enumeration 61
contev 222
contfp 229, 274
contfv 224, 229

**contlv** 225
**contvv** 220
Conversions 124–125
  from BR to CSG 130
  from octree to BR 130
  of BR to extended octree 130
  of CSG to decomposition model 93
  of CSG to bintree 98
  of CSG to BR 112
  of graphical model to solid model 30
  of sweeping model to BR 114
  of swept outlines to CSG 130
  to exhaustive model 60
Copying, of GWB solids 324
  undoing 324
**cross** 374
CSG 81–97, 124, 126
  as an interface to GWB 326
  for conversion to BR 112
  in a distributed modeler 128
  in a hybrid modeler 126
CSG tree 81–83, 126, 354
  in a BR modeler 126
  pruning of 98
**cut** 257
**cuta** 294
**cutb** 294
**cyl** 204

**del** 177, 198
**delcylinder** 348
**delhe** 181
**dellist** 177, 198, 259
Design transactions 126
  implementation 309
  integrity of 302
DF-representation 71
**displaysolid** 340
**dist** 242
Distributed modelers 128
Divergence theorem 116
Divide and conquer 85
**domovefac** 310
**dosetopgenerate** 275
**dot** 373
**dovertexonface** 276
Drafting system 22, 25–26
  on top of GWB 344
**dropcoord** 218, 229
Dual, of a polyhedron 108
Duality 47, 140, 150, 174

Eastman, C. M. 120
Encarnação, J. 11
**EndTransaction** 311
**erasesolid** 340
Euler characteristic 43–48, 140
Euler operators 139–160
  and the Euler-Poincaré formula 156
  completeness of 159
  implementation of 175–197
  modeling systems with 216
  reading from a file of 320
  soundness of 159
  undoing of 304
  validity of 159
  verbosity of 323
Euler-Poincaré formula 45–47, 154
Extended octrees 72, 129

Face adjacency graph 174
**faceeq** 219, 229
Faceting model 128
  representation of 347
Faux, I.D. 26
Feature extraction 130, 352
FEM 73, 75
**fface** 188
**fhe** 188

Gargantini, I. 71

Genus, altering of 144
definition of 46
evaluation of 324
GEOMAP-III 363
Geometric index 72, 129, 353–362
Geometric search 72, 352–362
GEOMOD 128, 363
**getmaxnames** 201
**getneighborhood** 247
**getnextnulledge** 255
**getsolid** 188
GKS 11
**glue** 209
GMSOLID 91, 126
Gouraud, H. 350
Graphical metafile 104
Graphical model 5, 13–28
as an interface to BR 114
problems of 29
Graphical symbols 19
Guibas, L. 57

Half-edge data structure 161–174, 323
Half-spaces 77-81
processing of 88
representation of 79, 347
Hanrahan, P. 30, 57
Henle, M. 57, 375
Henrion, M. 120
Hidden line removal, for BR 104, 116
for CSG 91
for graphical models 29
Hidden surface removal, for BR 116
for CSG 92
for octrees 68
for CSG of polyhedral primitives 128
Hierarchic models 354
Hierarchical enclosures 354
for boundary evaluation 354
for ray casting 354, 364
for set operations 356
Homogeneous coordinates 13, 365

Identification, of edges 38, 111, 163, 165
of plane models 41
of vertices 40
Image processing 59, 61, 75
Image recognition 352
Incremental boundary evaluation 91, 126
**insertnulledges** 252
**int2ee** 227
Integral properties 121
of BR 116
of CSG 93
of CSG by ray casting 93
of octrees 68
of polyhedral models 117
of GWB solids 226
Integrity 7
of boundary representations 110
of design transactions 126, 302
of graphical models 17
of hybrid modelers 125
of plane models 41, 145
of sweeping 114
Interrupt mechanism 301, 324
**intrev** 222
Invariance theorem 43, 45, 140
Inverse boundary evaluation 124, 130
Inversion algorithm 136, 315–323, 327
complexity of 323
**invert** 318, 327
**iread** 322
**isloose** 255
**isloosea** 294

islooseb 294

join 257, 261

kef 150, 196
kemr 151, 196
kev 149, 195
kfmrh 151, 197
Klein bottle 42
Koistinen, P. 72
Kunii, T. L. 130
kvfs 148, 194

Lamina 152, 200
    set operations of 300
larea 230
laringmv 231, 255
Lee, Y. T. 121
Lin, V. C. 27
Linear octrees 71
lkef 192
    undoable version of 308
lkemr 187, 192
lkev 191
lkfmrh 193
lkvfs 190
lmef 185, 191
lmekr 193
lmev 184, 190
    undoable version of 306
lmfkrh 193
loopglue 212, 294
lringmv 194

main 343
maketop 341
makeview 372
Markowsky, G. 30
match 212, 218
mate 171, 183
matident 373
matmult 373
matrotat 373
mattrans 373
MEDUSA 27, 128
mef 149, 196
mekr 151, 197
merge 210
mev 149, 189, 195
mfkrh 152, 197
MODIF 115, 216, 363
Mortenson, M. 26
movefac 260
    undoable version of 310
Mozart, W. A. 11
mvfs 148, 182, 190
Möbius' rule 43, 111
Möbius strip 38

nbrpreproc 282
neighbor 252
new 176
newcylinder 348
Newell, M. 217
nextedge 318
normalize 374
Null edges 213, 238
Null object detection 98
numget 337, 338

Object model 5
Octrees 63–70, 75, 354
    extended 129
    linearized 71
Orientability 42, 45, 57, 111, 140
Orientation, of a solid 294, 299
    of edges 106
    of polygons 104, 111

padl-1 91, 98, 126
padl-2 91, 98, 126
PADL-project 57, 98
Parametric design 25, 27
Parametric surfaces 125, 363
Perspective transformation 67

Phong, B. T. 350
Plane models 57, 110, 160, 161
  definition of 36–41
  manipulation of 139, 316
  orientability of 42
  representation of 163
Point-in-Polygon problem 221
Polyhedral model 7, 29, 101, 103, 128, 363
  in a hybrid modeler 126
  in an extended octree 129
  integral properties of 117
  processing of 116
**poppart** 341
Pratt, M.J. 26
Primitive instancing 52, 65
**processedge** 274
Pruning, of CSG trees 98
  of hierarchical models 354
**pushpart** 341

Quadric surfaces 78, 347
Quadtrees 63, 75

Ray casting 92, 364
  for BR 116
  using hierarchical enclosures for 354
  with EXCELL 362
Ray tracing 92
**readop** 320
**reclassifyonedges** 250
**reclassifyonsectors** 249
Regular set, definition of 33
Regular subdivision 59
Regularized set operations 84, 269
**remedg** 319, 324
**remfac** 321
Requicha, A. A. G. 57, 121, 298
**revert** 298, 299
  undoing 307
Ring 165, 231
  defnition 147
  moving a 197
**ringmv** 197, 231
  undoing 307
**rmsolid** 340
Rogers, D.F. 26
ROMULUS 115, 126, 216
Rotation transformation 366
  of a GWB solid 231
Roth, S. D. 98
Rounding 115
*r*-set, definition of 33
**rsweep** 207, 215

Samet, H. 72, 98
Scaling transformation 367
**scanjoin** 295
Scan-line algorithms 116, 121
Schlechtendahl, E.G 11
**sctrwithin** 284
Sector, definition of 238
**sectoroverlap** 285, 299
**sectortest** 284
**separ1** 293
**separ2** 293
Set membership classification 85
Set operations 81, 84, 125, 348, 353
  complexity of 353
  for 2-d profiles 114, 264
  for BR 112
  for exhaustive enumeration 61
  for GWB 263–300
  for half-spaces 78
  for laminae 300
  for linear octrees 71
  for octrees 67–68
  using hierarchical enclosures for 356
  with EXCELL 359
**setop** 272
**setopclassify** 277

**setopconnect** 296
**setopfinish** 297
**setopgenerate** 274
**setopgetneighborhood** 281
**setsurf** 348
**sgetnextnulledge** 294
Shape model 5
Shell, definition of 46
**sinsertnulledges** 292
Skeletal plane model 141, 147, 148
Slicing 261
**smef** 196
**smekr** 197
**smev** 195
**sortnulledges** 255
Soundness, of Euler operators 159
of plane model operations 145
**split** 241
**splitclassify** 246
**splitconnect** 256
**splitfinish** 259
**splitgenerate** 243
Splitting 233–262
**srecledges** 289
**sreclsectors** 287
**srotat** 231
Stolfi, J. 57
Storage allocation 175
**strans** 231, 324
**stransf** 231
**strget** 337, 338
Strut, definition of 195
Surface model 5, 78, 363
**svolume** 228, 231
**sweep** 203
Sweeping 96, 114, 336, 350
conversion to CSG 130
ray casting of 97
rotational 97, 202, 207, 216, 364
rotational sweeping of closed figures 213, 215, 216
translational 96, 200, 203, 364
undoing translational sweeping 216
validity of 114
SYNTHAVISION 93, 98

Tamminen, M. 72, 98
Tilove, R. 91, 98
TIPS 61, 80, 96, 357
Tomography 60
**torus** 213
Translation transformation 366
of a GWB solid 231

Unambiguity 50–51
of boundary representations 119
of cell decompositions 74
of CSG 97
of exhaustive enumeration 62
of half-space models 80
of octrees 69
of primitive instancing 55
Undoing 348, 350
of copying 324
**undoop** 312
**UndoTransaction** 311
Uniqueness 50–51
of boundary representations 119
of cell decompositions 74
of CSG 97
of exhaustive enumeration 62
of half-space models 80
of octrees 69
of primitive instancing 55
**unsweep** 216, 300
**updmaxnames** 271
undoing 309
User interface 10, 27
for GWB 325–343
to a CSG model 80

Validity 49, 51
of boundary representations 119
of cell decompositions 74
of CSG 97
of Euler operator sequences 159
of exhaustive enumeration 61, 62
of half-space models 80
of primitive instancing 55
of sweeping 114
**veccopy** 374
**vecequal** 374
**vecminus** 374
**vecmult** 373
**vecnull** 374
**vecplus** 374
Vertex neighborhood 238
classification of 238, 277
Viewing operation 368
Visual modeler 128
VLSI 62, 70, 97
Voelcker, H. B. 298
Vossler, D. L. 130
**vtxvtxclassify** 279

Wesley, M. A. 30
Winged-edge data structure 106, 135, 147, 163
Wire 202
Wire frame problem 89
Wire frames 6
conversion to a polyhedral model of 30
Woo, T. 130
Woodwark, J. R. 26, 98
**writeop** 316, 320

YACC 339
**yylex** 338
**yyparse** 340

Z-buffer algorithm 116

## ABOUT THE AUTHOR

**Martti Mäntylä** is currently a full Professor of Computing Technology with the Laboratory of Information Processing Science at the Helsinki University of Technology, where he received his M.Sc and his Dr.Tech. in 1979 and 1983. In 1983-1984 he was a Visiting Scholar with the Computer Systems Laboratory at the Stanford University. Dr. Mäntylä's research interests include computer applications in engineering, CAD, computer graphics, user interfaces, and data base management. He has published over 30 papers in these areas and is best known for his work on the use of Euler operators as a basis of solid modeling. Currently he is heading a research team working on a distributed mechanical CAD system based on advanced solid modeling techniques, interactive raster graphics, and networked engineering workstation technology. Dr. Mäntylä is a member of the ACM (and the ACM Special Interest Group on Graphics, ACM/SIGGRAPH), the IEEE Computer Society, and the Eurographics Association. He is also the representative of the Helsinki University of Technology in the Geometric Modeling Program of CAM-I.

## ABOUT THE BOOK

*An Introduction to Solid Modeling* aims at the construction and processing of "informationally complete" representations of three-dimensional solid objects—a technology fundamentally important in increasing the capability of applications such as computer-aided design, robotics, and three-dimensional computer animation.

This book is the first text solely devoted to solid modeling. In addition to giving a good overview of the current work in the whole rapidly changing area (Part I), the book includes a bottom-up description of a simple solid modeling system (Part II), including tested and working programs presented in the C programming language. Together with numerous programmming assignments presented as problems, the codes sum up to a simple yet fully functional solid modeler that can easily be interfaced with graphics facilities available, for example, in modern personal computers.

The book is valuable not only for those actually involved in the design and implementation of modeling systems, but also to those who need qualitative knowledge on the solid modeling technology as a basis of selecting a commercial system for an application or for developing new applications on the top of an existing system, for example. Professionals working in the fields mentioned above, or just interested in three-dimensional computer graphics in general, will find the book helpful in deepening their professional skills. For educators, the book constitutes ideal material for a half-semester advanced course on three-dimensional graphics.

ISBN 0-88175-108-1